Escola d'Enginyeria de Telecomunicació i Aeroespacial de Castelldefels
UNIVERSITAT POLITÈCNICA DE CATALUNYA

MASTER THESIS

TITLE: Study, design and implementation of WebRTC for a real-time multimedia messaging application

MASTER DEGREE: Master in Science in Telecommunication Engineering & Management

AUTHOR: Xavier Lagunas Calpe

DIRECTOR: Juan López Rubio

DATE: February, 17th 2017

Tirol: Study, design and implementation of WebRTC for a real-time multimedia messaging application

Autor: Xavier Lagunas Calpe

Director: Juan López Rubio

Data: 17 de Febrer de 2017

Resum

Les xarxes socials ja han deixat de ser un fenomen, i són, avui dia, no tan sols una realitat sinó quelcom indispensable. Durant el seu creixement i consolidació internet ha patit una gran transformació degut al tipus de continguts més demandats. Compartir imatges, vídeos o fins i tot establir una trucada amb una altra persona són tasques que un usuari normal pot fer diverses vegades al dia.

Aquesta transició només pot anar de la mà de noves tecnologies que no només simplifiquin aquestes tasques sinó que, degut a la irrupció dels smartphones, a més treballin sota unes condicions determinades com consum de dades i bateria moderat.

Les videoconferències a la xarxa, i en general, l'enviament de fluxos de data multimèdia ha anat sempre lligat de la mà de tecnologies de software tancat, com ara Macromedia Flash que requerien per part de l'usuari consumidor de la instal·lació d'un connector al seu navegador web.

Sota aquestes premisses, aquest projecte es centra en la investigació de WebRTC com a tecnologia capaç de realitzar vídeo conferencies entre usuaris sense la necessitat de cap connector al navegador.

Per contrastar els coneixements obtinguts a l'estudi es proposa addicionalment, el disseny, arquitectura i implementació d'una aplicació que sigui capaç de dur a terme aquesta tasca.

Title: Study, design and implementation of WebRTC for a real-time multimedia messaging application

Author: Xavier Lagunas Calpe

Director: Juan López Rubio

Date: February, 17th 2017

Overview

Social networks are no longer a phenomenon; nowadays it is not that they are a reality but have become something indispensable. During its growth and consolidations period internet has suffered a great transformation due to the new kind of most demanded content. Sharing images, videos or even making calls with another user are tasks that an average user would make several times a day.

This transition could only happen thanks to new technologies that not only simplify those tasks but, due to handheld devices' irruption, would work successfully under reasonable data and battery consumption rates.

Videoconferences over the network and multimedia data streams in general have always gone hand in hand of closed software products like Macromedia Flash, for instance, that required of a plugin installation on the browser by the end user.

Under those premises, this project will focus on the investigation of WebRTC as a technology capable of successfully achieving videoconferences between users without the need of any browser plugin.

In order to verify the knowledge gathered through the study of the technology, the design, architecture and implementation of an application capable of doing so will be proposed.

A Raquel: per recolzar-me sempre i fer que aquest treball arribés a bon port.

A Juan López: per la seva ajuda i bona disposició tot i la llarga durada del projecte.

A la meva família: per portar-me fins on sóc i per seguir ser sempre al meu costat.

ÍNDEX

FIGURES

INTRODUCTION

Nowadays seems very unlikely that a new technology could emerge with such a force that could displace most of the actual leaders of a given field. Although this sentence may apply in almost all the scenarios, there's one, the Internet, where this truth doesn't hold.

WebRTC is, very likely, an unknown topic for most of the people who surf the Internet every day, not only the consumers but also for most of the producers of its content.

It started as a means for providing any web page with an easy way to get multimedia capabilities as well as to *talk* to other clients on *real*-time but it has proven to be much, much more powerful than that; actually is powering several of the most used applications on smartphones such as WhatsApp, Google Hangouts or SnapChat, amongst others, which means that those big companies (maybe except Google because started the project) had had to change or refactor their products to make room for WebRTC in some or other way.

This project will take a look at this technology and will try to explain the main pieces of the whole WebRTC framework, and will do so by splitting into two halves this whole document.

The first part, comprised for the 1st and 2nd chapters, will aim at explaining all theoretical concepts which will provide the understanding of what this technology does and how can it make it.

The second part, comprised for the 3rd and 4th chapters, will consist of explaining how to build a real use case of this technology as well as showing the results; this use case will be, simply put, to create a web application that could handle videoconference from any open sourced browser provided with the WebRTC framework. This will be the main challenge of the thesis and if this proves to be feasible, then the next step will be to keep going further with this technology to see how far can one reach with it.

Finally, in the 5th chapter, with all the knowledge gathered through the course of the project will try to analyse, as a conclusion, if it is reasonable to create a white-labelled application relying only on WebRTC capable of mimicking all the functionalities of any of the top messaging applications mentioned above.

CHAPTER 1. WEBRTC

WebRTC is a dense topic that involves many technical concepts. It is bundled to work upon high security standards from the very beginning and relies on a wide range of well-known protocols like, for instance, RTP, RSTP, SDP, etc. It is designed to take care of most of the work so that the user, in this case the developer, has to focus only on how it want to leverage its power rather than having to struggle with small implementation quirks.

This introductory chapter is intended to introduce WebRTC: What is WebRTC, what can do and give a high level overview of what capabilities can bring from a developer point of view.

At the end of the chapter, the main API and uses of WebRTC will have been explained as well as the roles and responsibilities of each of its subcomponents.

1.1. What is WebRTC

Web Real-Time Communication, WebRTC, is a standard that defines a collection of communication protocols and application programming interfaces that enable real-time communication over peer-to-peer connections directly from the browser.

This allows web browsers and other clients to not only request resources from backend servers, but also real-time information from other browsers or clients of other users. This enables applications like video conferencing, file transfer, chat, or desktop sharing without the need of either internal or external plugins.

In this definition there are three key aspects: Peer-to-Peer architecture, lack of need for external plugins and being a work in progress standard. Those are the foundations of WebRTC and are the reason of its increasing adoption.

On one hand, the fact that the whole architecture is peer-to-peer based gives a great scalability to any given solution based on WebRTC. The reason is quite simple; multimedia streams demand a huge amount of bandwidth (this is directly related to the quality -which encoding algorithms, which format or if it allows losses- of a given data stream), and the fact that WebRTC embraces a decentralised architecture means that there will not be trunk nodes that will have to handle all the traffic generated by the simultaneous connections and as a result, compared to a classic client-server architecture, a higher number of simultaneous connections can be achieved since it is not limited by the maximum throughput one single machine can handle.

The second and third major benefits are quite related with each other; on one hand there's the fact that WebRTC is an open standard and it leans on other standards, like HTML5, and other mature protocols such as RTP, RTSP, SDP.

That means any vendor has the opportunity to support it in its codebase for free, taking the benefit of the improvements that the community achieves. This is a winning point because it favours the interoperability of any given platform built using that technology at close to zero development time overhead.

As said before, the other great benefit of adopting WebRTC is it doesn't need a plugin. Not so long ago, almost every multimedia stream that one could encounter would be relying on Macromedia Flash technology. Actually Flash was quite the opposite of WebRTC; it was, and actually is, maintained by one company, Adobe, and also was closed source (a binary file was generated and that made that all browsers needed a custom plugin in order to be able to playback those streams). As it is known, those plugins had lots of vulnerabilities, fact that forced most of the major browser providers, starting with Apple's Safari, to slowly transition towards disabling it by default.

At that point it seemed logical that some open standard alternative should step in and provide a valid solution for the whole web environment, and that's why WebRTC was made.

1.2. A brief history of WebRTC

WebRTC story starts on May 2011. This was the date when Google released an open source project for browser-based real-time communications known as WebRTC. This project was part of the source code that was running Hangouts, Google's videoconference web application.

Starting with that initial library, efforts where put to standardise the key protocols and browser APIs involved (mostly Chrome-related since it was and still is Google's own browser) by the IETF in the W3C.

Currently, the W3C draft of WebRTC is a work in progress with advanced implementations for Chrome and Firefox browsers. The API is based on preliminary work done in the Web Hypertext Application Technology Working Group (WHATWG), whose founders where Apple, Mozilla Foundation and Opera software on 2004. It was called the PeerConnection API, and a pre-standard concept implementation was created at Ericsson Labs.

The main interests of the Web Real-Time Communications Working Group revolves around two main aspects:

The first one is to keep track of the changes in the parallel RTCWEB group at IETF. That is, check that the corresponding APIs on WebRTC matches the set

of protocols that define real-time communications on Web browsers. This involves detecting and avoiding privacy problems when exposing local streams; for example, requiring explicit permission of the user when requesting a webcam or audio stream.

Regarding the two initial implementations, Firefox and Chrome browsers, first early additions of their implementations could be found on their beta versions:

- Chrome desktop version 18.0.1008.
- Firefox browser version 17

Initially, those implementations covered only GetUserMedia and PeerConnection APIs. This will be covered in full detail later but, for now, suffices to say that GetUserMedia API is in charge of requesting and obtaining local video or audio streams whereas PeerConnection API is in charge of connecting client parties with each other thus allowing a full multimedia content exchange.

It was on February 19th, 2013 when Firefox added support for PeerConnection and DataChannel APIs into the browser stable version. On the other hand, Chrome made it available on its stable channel on version M27 on March 25th, 2013.

As can be imagined, although both vendors where implementing the same WebRTC draft, as it comprises a huge list of technologies, each of them prioritized different features on their roadmaps. That lead to some incompatibilities between them; for instance, the exchanged information on Chrome navigator was not being ciphered while it was on Firefox. That's why although both navigators where using the same set of technologies, they weren't able to talk with each other for the moment.

That changed on one of the biggest initial milestones of the project. This was the sign that WebRTC was starting to become a serious contender.

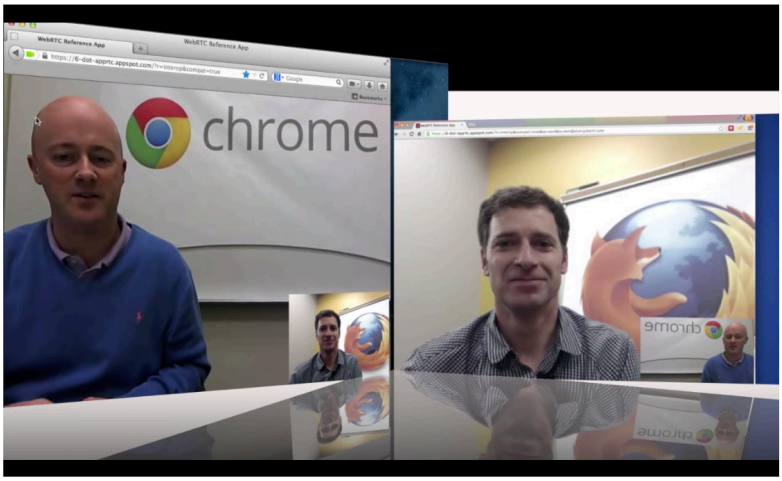

Fig. 1.1 Chrome and Firefox interoperability demonstration

After this, Opera's browser added support for WebRTC APIs as well on its 18th version. From that moment all three browsers where able to call each other seamlessly

On the other side of the market, some of the companies based on proprietary software for their browsers have done some moves towards WebRTC. On one hand, Microsoft developed ORTC; an API which is on-the-wire compatible with WebRTC 1.0. That means that although it's a custom wrapper around WebRTC framework, it should be able to interoperate with standard WebRTC.

Finally, the other company leading on the proprietary software side, Apple, showed initially no interest in this technology arguing that they already had Facetime technology although it has been reported that they have recently started working towards adding WebRTC on Safari's browser. So far there's no advance in that matter for the end user or the development releases of the browser so it sounds more like a long-term feature rather than something close to happen soon.

On other order of things, as a second great milestone regarding WebRTC project, it is important to remark that nowadays, mobile traffic has surpassed desktop traffic therefore WebRTC desktop implementations have also been ported to their sibling mobile versions for Chrome, Firefox and Opera browsers on Android and iOS platform in a move to enlarge its market.

The following section will verse about the state of the art of the technology as long as what are exactly the technologies bundled in WebRTC

1.3. State of the art

As stated before, the fact that WebRTC is currently an unfinished draft, most of its features are still pending to be totally implemented and the fact that each vendor has its own roadmap implies that although there is common consensus about the core features that every browser has to implement (primarily video, audio and interconnection capabilities), there is not any pre-defined guidance about the order in which the rest of the specifications have to be taken into consideration, that means that until the project is far more mature than it is now, only the core capabilities will be able to be used seamlessly between platforms.

Regarding the state of the art of the technology, Fig. 1.2, shows each of the features or technologies that define WebRTC and its current implementation status by browser.

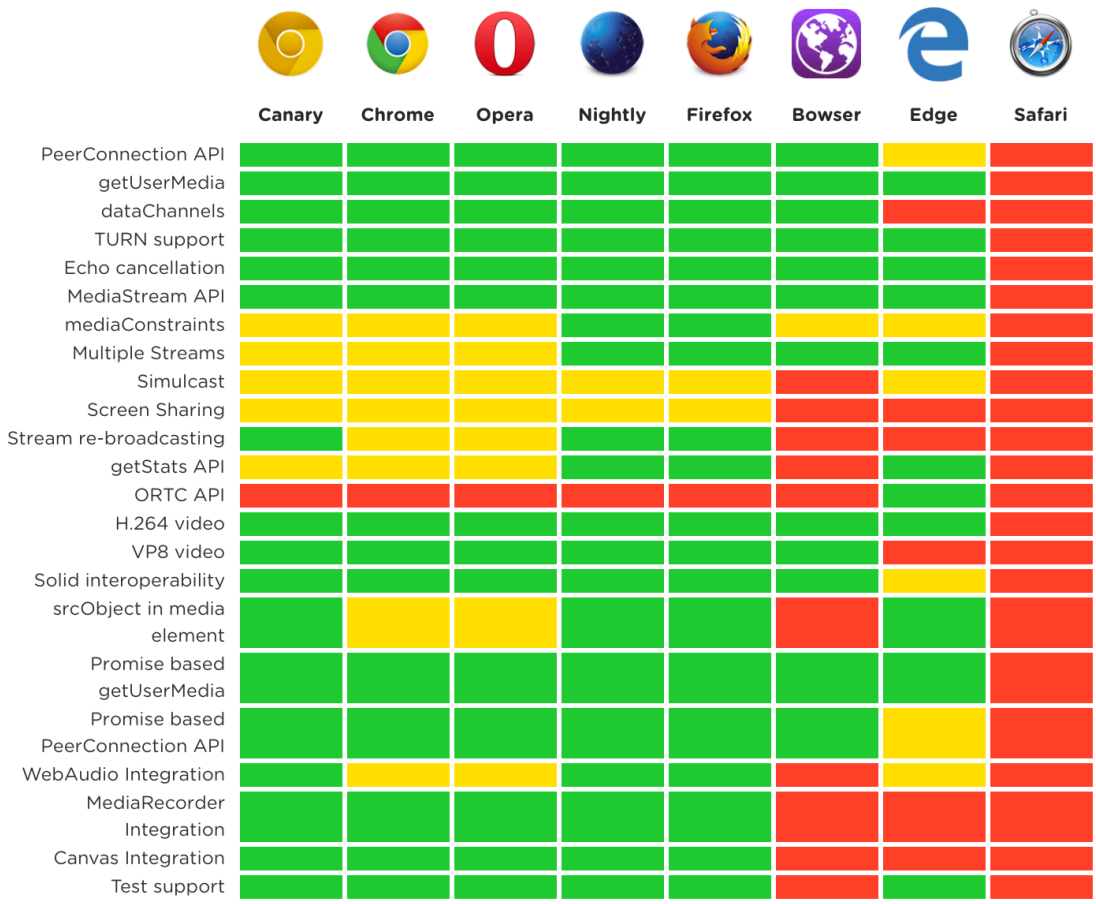

Fig. 1.2 WebRTC comparison table

In order to understand the table, the green fields mean that the specification is fully implemented, yellow ones mean that there is partial or that it's not fully adhering to the specification and finally, red ones mean that there's no implementation that cover that area yet.

With a quick look at the table, several things can be spotted easily:

- Safari (Apple's browser) still has not implemented any of the APIs that WebRTC provides which means that real interoperability is not possible at this point provided how much market share this browser has.

- Microsoft is developing its own standard, ORTC. They state that it is a wrapper around WebRTC and that will provide an easier API for the developers to interact with but, as can be seen in the table, none of the other browsers are implementing it.

- Most of the open-sourced browsers (or at least those whose engine is open-sourced, like chromium for chrome) have already implemented most of the APIs being cross-compatible between them.

Next chapter will be an in deep jump into the APIs and components that have been used in the development of the proof of concept application.

CHAPTER 2. Acquiring and broadcasting multimedia streams with WebRTC

As explained in the previous chapter, WebRTC is a framework with lots of different functionalities and services, which are provided by a set of APIs. Of all its functionalities, this project will explore those that enable the possibility to achieve a videoconference successfully.

At first glance, it seems reasonable to break a videoconference in three independent chunks, first one, acquire both local audio and video streams from the user, and after that, connect with the other end, where the other user will do the same and, after that, cross-share those streams with each other.

Finally once both ends have all the streams (audio and video) for both parties, they have to be played back, in this case in a web page, so it will be done inside some HTML5 <video> tag.

WebRTC provides two APIs that help fulfilling these requirements, GetUserMedia (or MediaStream) and PeerConnection APIs. Guessing by its names seems clear that the first one will provide means to obtain video/audio streams and the second one will help connecting both endpoints or users.

2.1. Acquiring media streams through MediaStream API

This is the API that provides the user with the streams that he wants to acquire. It's the simplest and most elementary API on WebRTC. The current implementation status is focused only on providing webcam and microphone hardware access through an interface called MediaStream; each MediaStream is comprised of 1 or more streams or MediaStreamTracks from different types.

This last point is related to the synchronicity needed between the different streams, for instance, if a user wants to acquire one type of either audio or video streams, there is not any synchronisation problems to be considered, but on the other hand, if a user wants to perform a videoconference, it is important than both audio and video streams are synchronized. It seems obvious that even though those streams are from different hardware sources, they are related and to solve that the MediaStream API bundles both streams together inside the returned MediaStream object.

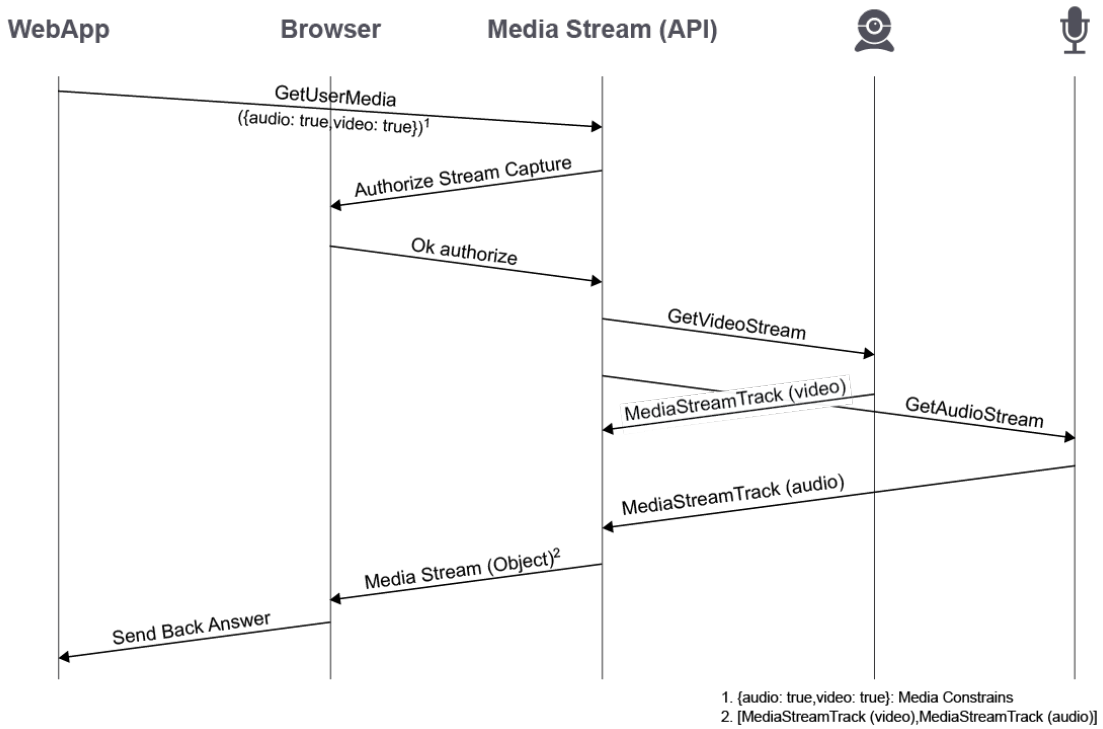

Fig. 2.1 GetUserMediaAPi internal calls for videoconference scenario

The graph above shows the flow a request takes starting at the point where access to one or several hardware streams is requested until the point it is granted. There are 3 important aspects highlighted in the graph that need to be looked more in depth: the internals of MediaStream, MediaConstraints and security validation.

2.1.1. MediaStream API internals

As told before, MediaStream API is the entry point from a developer point of view to the WebRTC framework. Starting from this call and obtaining one or several streams, will be able to manipulate them, send them, or modify them (for instance running some computer vision framework to work with a video stream or to apply audio effects to a given audio stream).

To start understanding this API, first step is to go to the core of its functionalities, that is, what is the main issue it addresses, and in this case is provide the user of media streams.

To do so, there needs to be an architectural abstraction that is able to handle all kinds of different media streams regardless of what source they come from, or what characteristics they have. This means that this API has to work exactly the same way either for Audio or Video streams, and to do so, as its been said earlier in this chapter, there is an abstraction of a stream called MediaStreamTrack. Below are the details of what this data structure holds.

```webidl
[Exposed=Window]
interface MediaStreamTrack : EventTarget {
    readonly attribute DOMString             kind;
    readonly attribute DOMString             id;
    readonly attribute DOMString             label;
             attribute boolean               enabled;
    readonly attribute boolean               muted;
             attribute EventHandler          onmute;
             attribute EventHandler          onunmute;
    readonly attribute MediaStreamTrackState readyState;
             attribute EventHandler          onended;
    MediaStreamTrack       clone();
    void                   stop();
    MediaTrackCapabilities getCapabilities();
    MediaTrackConstraints  getConstraints();
    MediaTrackSettings     getSettings();
    Promise<void>          applyConstraints(optional MediaTrackConstraints constraints);
             attribute EventHandler          onoverconstrained;
};
```

Fig. 2.2 MediaStreamTrack data structure according to W3C

Taking a look at the definition, first thing that comes to mind is that this interface inherits from EventTarget which is an interface implemented by objects that can receive events and may have listeners for them.

Reason is simple, this MediaStreamTrack is going to be having to react to different events like, pausing the stream whenever the user asks for it, or cleaning resources when there's some error that makes that the stream is no longer valid (for instance, when accidentally disconnecting a webcam it is needed), it is important to provide the developer of some ways to gracefully handle this cases so it could recover (maybe the preference is to switch from a videoconference to an audio call without needing to break the call).

Looking into the attributes and functions that this class exposes, an interesting fact is that most of the attributes are read-only; those are the ones that characterizes the stream, they give information about what kind of stream it is (video or audio), the id to uniquely identify them and the label field to give a brief description of what type of source it comes from, for example, internal microphone. The two other fields are muted, which means that there are no live samples of the stream and, finally, state, which is simply a flag to indicate if a stream is either live or ended.

On the other side, there are also read-only helper functions to characterize in more detail the given stream such as getCapabilities(), which returns a list of the exact features of the current media stream. Explanation is simple, once the user pass its MediaConstraints request, as will be shown later, there's no guarantee that the requested conditions are exactly met so, with this information, a user can understand what exactly is being delivered. Things like frame rate, width, height, volume, etc. The idea is that MediaConstraints should almost match 1-to-1 all this features, so that the user could fine-tune the stream as much as needed.

Other fields, simply are exposed for the internals of the API to work with, but are of no use for the developer or the web application.

Now that the smallest data structure that characterizes the streams have been explained, it is time to talk about how does the developer interact with the API.

It is, actually, quite simple. Once a valid request to GetUserMedia() is done, the returned value is an object of class MediaStream which holds the following structure:

WebIDL

```
[Exposed=Window,
 Constructor,
 Constructor(MediaStream stream),
 Constructor(sequence<MediaStreamTrack> tracks)]
interface MediaStream : EventTarget {
    readonly attribute DOMString    id;
    sequence<MediaStreamTrack> getAudioTracks();
    sequence<MediaStreamTrack> getVideoTracks();
    sequence<MediaStreamTrack> getTracks();
    MediaStreamTrack?          getTrackById(DOMString trackId);
    void                       addTrack(MediaStreamTrack track);
    void                       removeTrack(MediaStreamTrack track);
    MediaStream                clone();
    readonly attribute boolean      active;
             attribute EventHandler onaddtrack;
             attribute EventHandler onremovetrack;
};
```

Fig. 2.3 MediaStream interface according to W3C specification

Flow since the user request a stream until it is obtained is as follows:

1. Analyse media constraints and check how many different streams are requested and its specific requirements
2. Create a MediaStream object
3. For each of the type of streams stated in the Media Constraints JSON, check if there is some hardware or software sensor that matches the required stream, if there is no source that matches the mandatory requirements, send an error exception and exit, otherwise proceed with step 3.
4. Generate a MediaStreamTrack from the hardware/software's sensor that satisfies the specifications
5. Add this MediaStreamTrack to the MediaStream object
6. Once a track has been added, trigger the onaddtrack EventHandler, which is a callback function
7. Finally back the MediaStream object as a result to the requested call

At the end of the day, MediaStream object is just a wrapper that holds a list of MediaStreamTracks that work together, synchronously. It is bundled with utility functions to help make searching for a specific track faster. For instance it has a method that simply returns all the collected MediaStreamTracks without any distinction of type, getTracks(), while it has as well, utility methods to filter by audio or video tracks, getAudioTracks() and getVideoTracks() respectively, this filter is done taking into account to the read-only type attribute of the MediaStreamTrack.

Finally, there are two callback functions that are triggered whenever a control event occurs, such as adding or removing a track to help the user wire together a stream with its corresponding HTML5 tag (either <video> or <audio>) so that

it could start being played. This will be explained with more detail in the subsection called playing media streams through Blob-URLs

2.1.2. MediaConstraints

If the MediaStream API can provide any type of stream from a single API call, there has to be a way to address which kind of stream is being requested, that's what MediaConstraints are for.

The constraints parameter is at the end of the day a JSON object with two members: video and audio, describing the media types requested. Either or both must be specified.

The idea behind this is that if the browser cannot find all media tracks with the specified types that meet the constraints given, then an error will be thrown to notify the user.

MediaConstraints, in its simplest form look like this:

```
{ audio : true, video: true }
```

Whenever any media type is flagged as true means that the resulting MediaStream is required to have a MediaStreamTrack of that type in it.
The example above is, thus, the required for a videoconference but this is only the mandatory part of the MediaConstraints.

As stated into the introductory chapter, WebRTC was designed taking into account security as one of the main concerns, having said that, one of the main implications to this, is the fact that there's no access to the underlying hardware information, either cameras or microphones. That means that given the wide range of devices and different capabilities, some way to add some fine-grained constrains should be added.

```
{
    audio: true,
    video: { width: 1920, height: 1080 }
}
```

With the previous JSON snippet, the user is requesting a preferred resolution to work with. This is a blind guess, since there's no way that the API provides him with some of the available capabilities of the current devices he is using (actually there's not even any guarantee the computer or laptop has camera built-in), so in this scenario the framework chooses the resolution that could match in a better way the requested one but in any case this mean that if this resolution was not supported in the provided hardware then an error would be thrown.

Obviously, there are another scenarios where a given resolution is mandatory. There are some fields, like medical, where some minimal resolution quality must be met in order to be able to, for instance, perform some remote diagnostics. For this type of scenarios, there is another approach to ask for a required resolution:

With the addition of min, ideal and max keywords, the user is able to state a range of resolutions that could fit the scenario he is on.

```
{
    audio: true,
    video: {
        width: { min: 1920 },
        height: { min: 1080 }
    }
}
```

In this case, the user will meet the opposite behaviour to the previous one, in case that there's no resolution that matches this constraints, an error will be thrown and no stream will be returned.

Analogously, the use of max constraint will produce the same behaviour but on the opposite constraint. This could be useful when dealing with streams on low-end devices or on 4G scenarios, where sending high quality streams could lead to a bad user experience due to lack of fluidness in the videoconference or running out of user's 4G data plan.

Finally, to close this series of resolution specification examples, there's the case were a user can specify a range of resolutions as well as they use of the keyword ideal, where as one can imagine, states which is the preferred of the range. In case the ideal resolution can't be fulfilled, then the one that will be used will be the closest one to the desired:

```
{
    audio: true,
    video: {
        width: {min: 1024, ideal: 1280, max: 1920},
        height: {min: 776, ideal: 720, max: 1080}
    }
}
```

On the other side, there is a last consideration to be made. Nowadays, smartphones and tablets sport fully capable browsers, so as it is expected, those browsers are also supporting WebRTC. As a result, those handheld devices have certain specs that the standard laptop or desktop computers doesn't meet, for instance, they have 2 cameras, so there needs to be some constraint to deal with the ambiguity of which camera does the user refer to. Using the keywords *user* or *environment* inside the member *facingMode* will achieve the desired effect: user for front-facing and environment for the back camera.

```
{ audio: true, video: { facingMode: 'user' } }
```

Using the keyword exact to require in a similar way as min or max, so if the device doesn't have the required camera, it will fail.

```
{ audio: true, video: { facingMode: 'environment' } }
```

2.1.3. Security concerns

Regarding security, there are two different aspects to talk about regarding the MediaStream API.

First topic to discuss is relating the nature of WebRTC on browsers. Nowadays, browsers are powerful environments that can perform almost as native applications on a laptop or computer. The days were browser were only static HTML and CSS have gone and now the user finds himself at the era of the webapps, which blur the lines between laptops, desktop computers or handheld devices.

At the end of the day, front-end programming on web applications is being done in javascript, which is a runtime programming language that runs on the client's computer. Obviously it has lots of benefits, but what concerns in this chapter is security and in that matter there are things to take into account.

A user which browses the internet, in not aware of the scripts that every page is running as long as they don't update the UI and, generally, is not really interested in knowing how things work. If a webpage has a nice animation, it does not matter how it is done, the only thing that matters is the result, and dealing with the kind of contents that MediaStream API is handling, some measures had to be taken.

It is easy to imagine a scenario where this is a real thread. If no security is taken into account, suffices to think about a malicious webpage which taking advantage of this API silently requests the user to remotely turn on the webcam in order to spy him. How does the user know that someone is remotely tracking him? After all, the user only browsed through a webpage and he did not notice anything strange.

That scenario highlights why this API and WebRTC as a broad term had to be designed with security as one of the key pillars.

The solution to this scenario is really simple and it comes out of the box:

Fig. 2.4 Chrome browser asking user's permission to use some stream

Whenever a user lands on a web page which makes a request to use MediaStream API, he will be showed a popup in the navigation screen making him chose whether he wants to allow the use of the API and show any stream or not; only when the user answers the popup, the request finishes returning back its output to the API caller.

Chrome development team raised the second issue and all the other browsers are willing, if not already done, to follow the same restrictions. It is called Secure Context and MediaStream API has to comply with it.

As the web platform is extended to enable more useful and powerful applications, it becomes increasingly important to ensure that the features, which enable those applications, are enabled only in contexts that meet a minimum security level, accessing personal image/audio data is considered one of the most private actions that can be taken care of in a web so that's why it is considered that MediaStream API has to be called from a trusted source.

The most obvious of the requirements discussed here is that application code with access to sensitive or private data be delivered confidentially over authenticated channels that guarantee data integrity. Delivering code securely cannot ensure that an application will always meet a user's security and privacy requirements, but it is a necessary precondition so to sum up, if the web doesn't have a valid ssl certificate, it won't be able to make use of MediaStream API.

2.1.4. Playing media streams through Blob-URLs

This is the last section that covers the MediaStream API. It has been discussed how to acquire one stream, how to specify which type of stream and how to query and adjust the settings for the hardware sensor that the user wants to work with, as well as the security implications wrapped around WebRTC but so far, there is no way to actually see or hear any of the streams the API is providing.

To solve that, HTML5 comes to the rescue. The browser can convert into a so-called URL blob every MediaStream object returned by a successful call to the MediaStream API .

Browsers have a native method that does exactly that operation, window.URL.createObjectURL(), expects as a parameter the MediaStream object and outputs a String representing a URL in the following form:

```
blob:http://localhost/xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
```

Once this URL has been generated, it is only a matter of attaching this URL to either a <video> or <audio> src attribute and everything will be set up automatically for the user, last step would be to hit the play button to start streaming or, in case of videoconferencing apps, add the autoplay attribute so that once the streaming is hooked up, it will play automatically.

On a side note, here is where knowing that the streams are synchronous comes in handy. The fact that both audio and video streams are bundled together in the same MediaStream object allows the developer to only have to attach the generated URL blob to an HTML5 video tag and will take care of both, audio and video playing back synchronously.

If that part wouldn't have been thought of previously, developers but have been forced to attach both <audio> and <video> tags for each and every stream and having to handle any synchronization issues manually and although this could be overseen when working on local streams by neglecting the acquisition time latency, that would not be the case of remote streams, where jitter without a really good buffering technique could made this nearly impossible or on the other hand, do a videoconference impossible due to the delays.

2.2. Broadcasting multimedia streams through PeerConnection API

Next step in order to achieve a videoconference app is to share the previously obtained streams with the other end and, at same time, receive the other end's streams in order to play them back. That's exactly the role PeerConnection API plays inside the WebRTC framework.

PeerConnection's name itself gives a hint about some architectural design, Peer, stands for peer-to-peer architecture. This, as opposite of a classic, client-server architecture has some benefits like greater scalability as well as lower latency.

Greater scalability because there is not a central server, or server cluster, that has to deal with all the traffic generated for all the clients that are using the app simultaneously and lower latency because being WebRTC a framework targeting real-time communications between two parties, what makes more sense is try to search the faster route from one endpoint to the other without

having to reach a fixed central server. For instance, imagine a scenario where app servers where physically located in the US and two users from the same city, Italy for instance wanted to talk to each other. There is no point on having to route all the traffic through that US central node. It would we a waste of latency and resources for all the intermediate routers, key on this kind of applications.

In order to achieve this, PeerConnection API had to be designed in a way that could solve that problem; First challenge that rises is the fact that not having a well-known central server means that there is no place to interconnect different users, actually, there is no server role in the PeerConnection API.

The other challenge is about agreements; how does two end users agreed in what codecs, bitrate or resolution should they use? So if client A is using Firefox browser and user B is using Chrome (both using their default video codecs), how will each browser render the other browser's stream? One thing is receiving a stream and another is being able to successfully decode it.

To fix both problems, PeerConnection API relies in two protocols, first one JSEP or Javascript Session Establishment Protocol, and the second one is SDP, Session Description Protocol.

Next subsections will explain which problem addresses those protocols and how they solve it

Javascript Session Establishment Protocol

WebRTC's main aim is to provide tools for handling the media streams but does not say a thing regarding the signalling between endpoints in order to not be tied to any particular technology.

That is a smart decision because it is flexible enough to be able to work with old and matured signalling protocols, like for instance SIP, but at same time, perfectly fit as well as for tailored solutions for new applications, for instance websockets could be another solution upon which to create a custom protocol to satisfy some custom applications need.

At the end, the main point of this protocol is to be able to successfully exchange the multimedia session description, which specifies the necessary transport and media configuration information necessary for the other end to decode the data.

With these considerations in mind, JSEP removes the complexity almost entirely from the core signalling flow, which the browser handles making use of two interfaces: passing in local and remote session descriptions and interacting with the ICE state machine.

Interactive Connectivity Establishment (ICE) is a technique used to find ways for two computers to talk to each other as directly as possible in peer-to-peer networking. If a user wants to avoid communicating through a central server (which would slow down communication, and be expensive), but direct

communication between client applications on the Internet is very difficult to achieve due to network address translators (NATs), firewalls, and other network barriers that data packets have to sort on the way to be delivered to its destination.

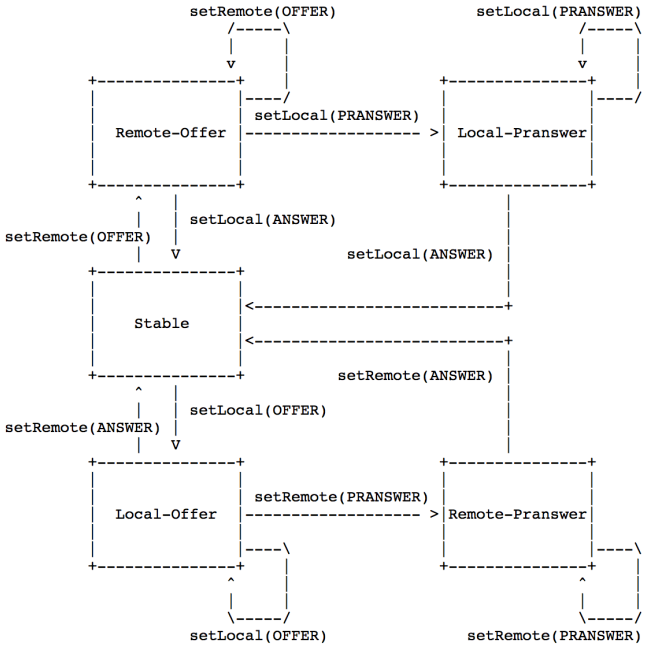

Fig. 2.5 JSEP Finite state Machine

JSEP's handling of session descriptions is simple and straightforward. Whenever an offer/answer exchange is needed, the initiating side creates an offer by calling a createOffer() API; the result is then used to set up its local configuration via the setLocalDescription() API.

The offer is then sent off to the remote side over the chosen signalling channel and will be received by the other endpoint. Upon receipt of that offer, the remote endpoint sets it up using the setRemoteDescription() API.

To complete the offer/answer exchange, the remote party uses the createAnswer() API to generate an appropriate answer, applies it using the setLocalDescription() API, and sends the answer back to the initiator over the signalling channel. When the initiator gets that answer, it sets it as well using the setRemoteDescription() API. At this point, the initial setup is complete. This process can be repeated for additional offer/answer exchanges.

So at the end of the day, JSEP is only in charge of exchanging each endpoint's media with the help of the createOffer/sendAnswer APIs:

If the user is the caller, or the initiator, will add the requested streams as his localStreams and will send an offer through the signalling channel.

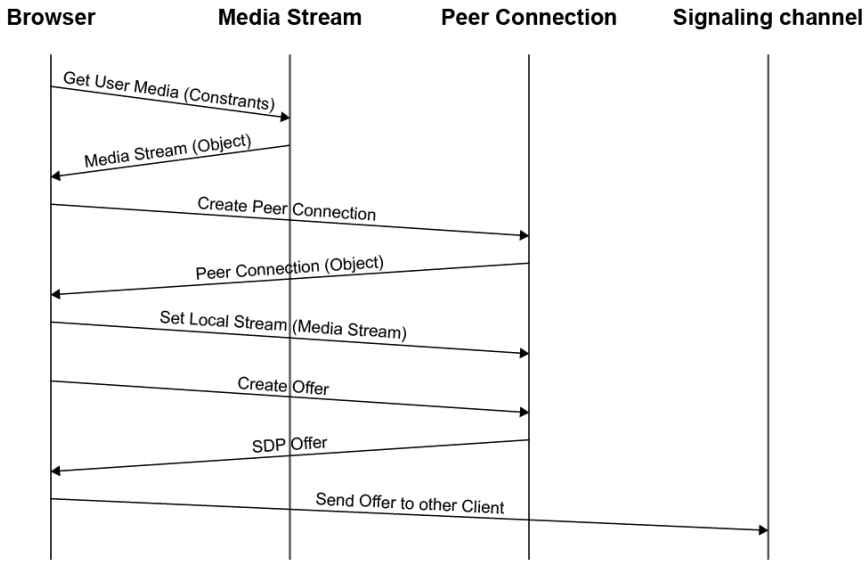

Fig. 2.6 Stream acquisition and offer sending

On the other side, the callee or receiver will get a message from the signalling protocol with the information of an SDP offer. At this moment, the caller will make a GetUserMedia call with the media constraints he wants or needs (they could be different than the ones that the caller required, after all it indicates how is the user sending data, not receiving it) to answer to the call, and once provided with the MediaStream object, will create a PeerConnection object.

Once he has that last object, he will proceed -the order is not important at this step- setting his localStreams with the MediaStream he has been granted and will add the remoteStream with the SDP information provided by the signalling message.

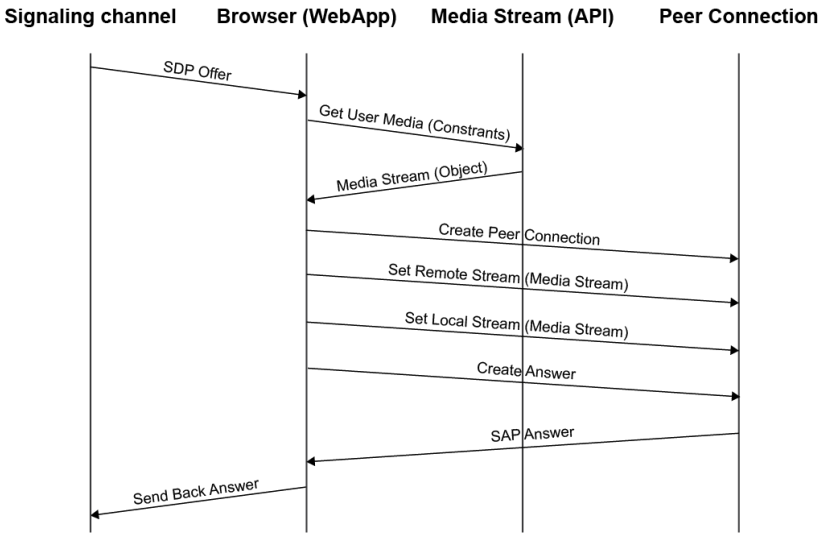

Fig. 2.7 Offer reception, stream creation and answer sending

At this point, the callee, is ready to share his streaming information to the caller, so he, through the createAnswer API call, generates his SDP information which will be sent back to the user through the signalling channel.

At this point, the callee has the information of both streams and is ready to send his to the other party.

On the other end, once the caller receives the signalling message with the answer, will use that SDP session to make a call to setRemoteStream with that session and will be finished as well.

Finally both users have all the information they need about each other and the session can, theoretically, begin.

CHAPTER 3. Designing and architecting a WebRTC-based messaging Web application

Previously, the first 2 chapters have explained the theoretical aspects regarding WebRTC and all its capabilities but as stated in the introduction, this second part of the document will deal with how to take advantage of those concepts that have been showed in order to successfully build a videoconference application.

It has also been explained that the main goal is to develop a web application solution that leverages the power of WebRTC and this chapter will explain the foundations of the application that has been developed.

This chapter will focus on two different things; on one hand, it will describe what will the application do by splitting it into different use cases so it is easier to understand. Afterwards, the second part will deal with designing the application architecture that will support the solution. The detailed implementation details will be explained on the next chapter

3.1. Defining a videoconference web application

First step in order to architect some application is having a clear idea of what is it that wants to be achieved. In this case it is quite simple; the aim is to create a web application that works well on handheld and desktop devices, that does not require of any extra plugins for the user and with which he can make videoconferences or audio calls.

Even though the main focus of the application is to be able to do videoconferencing, there are a lot of things that are not explicitly related with that matter but that must be done in order to have a successful application. For instance, the user should only be able to call contacts he trusts and that are previously authorized by himself to be part of the roster, or list of contacts, and on top of that, due to the nature of the platform that is being used (a web browser), calls can only be made between 2 users that are connected at same time, and that means that there has to be a way to handle and track that scenario.

On other order of things, a user should also have the chance to schedule a videoconference with another user in the event that the latter is no present at the time he wants to call him.

As nice to have features, there should be the possibility to make multi-conference calls and share rich-media content between the users inside of a call, from text to imagine or binary files.

Those are the initial requirements that the application should cover which are the standard ones in any messaging application such as Skype, and that is exactly one of the goals of this thesis, to determine if a white-labelled

messaging application can be done sporting the functionalities of the most downloaded communication applications.

3.2. Defining the use cases for a videoconference web application

This section will define the different use cases found on the application. There are several areas those use cases can be grouped by, according to the area of their functionality. The different areas are: authorization and authentication, videoconference, roster and meeting scheduling.

3.2.1. Authorization and authentication related uses cases

Authorization and authentication use cases are those who are related with providing access to the platform in a customised way. Authorization could mean who can or cannot access to the application or also a means of providing scoped access to given areas of the app.

Authentication on the other hand is the ability to identify a given user. There is no need to state how important that is; without a way of identifying users there's no way to customise content, monetize and app, or keep usage statistics.

Inside this category, there are two different use cases: registration and login.

Regarding registration a user has to be able to create an account on the platform. This account has to provide user details so that later on this user can be searched and identified uniquely. The fields that are going to be required are, username, full name and profile image.

Regarding login, there is really not much to say, it is the entrance on the application for a user. It has to provide a way of assessing that a user is who he blames he is, so there is the need of a password secure enough to prove it.

3.2.2. Videoconference related use cases

This is the core functionality of the app. There are a lot of actions regarding the videoconference section, so the better way to put them together is through a list:

1. A user has to be able to make a videoconference call.
2. A user has to be able to accept a videoconference call.
3. A user has to be able to reject a videoconference call.
4. A video or voice call will be rejected if the callee does not answer in less than 20 seconds.
5. A user has to be able to add more users to a running conference.
6. A user has to be able to finish a videoconference.
7. A videoconference room has to be able to resize or adjust to fit all the participants on the screen on a responsible way.

8. A videoconference call has to keep running until only one participant is in the room.
9. A user has to be able to chat with the other users on the same call.
10. A user has to be able to send binary data to the users on the same call.
11. A user has to be able to send picture to the users on the same call.

First 4 points on the list are related to two or more users agreeing to have a call between them. The reason is obvious, although the app could technically have the ability to avoid that point, there are privacy and security implications that make this the starting point of every call, only if both parties agreed to have a conversation should go further, otherwise nothing will happen and both users will return to their previous state.

Points 5 to 8 gravitate towards another concept as well, multi-conference. Although it is clear that a simple call between 2 users ends when any of them leaves or hangs up, that's not the case on videoconference. In this case, and taking into account that the app is rendered in a browser, it should have to be able to resize each of the streams in the screen so they all fit nicely. That implies that the calls have to be dynamic and not only set up from the beginning, making space for more users through the course of the same.

Finally, points 9 to 11 are support functions to make easier the communications between end users. It's important to understand that due to the nature of WebRTC, those files are not stored in any server but sent directly to all the parties. That makes the app avoid privacy-related security problems. The idea behind this last point is to make the application not only useful for maintaining a conversation but to make it a complete collaborative workstation area.

3.2.3. Roster related use cases

Every user should be only reachable to the users that he knows and trusts. This will be the basis of the communication in the application. Every user will have a list of contacts to call and he will only be able to call those who are online. That implies that there has to be a mechanism to flag all the users that are available, connected, at the time the user is.

On the other hand, a discovery-acceptance process is needed as well; one user searches for a contact and, once found, sends him a friendship request. After that, in the other end, is the requested user who decides whether to accept or decline the offering.

Although this sounds trivial, truth is, there are certain things that have to be taken into consideration and most of them are related to the fact that this mutual acceptance of friendship could change with time.

There are two approaches to support this appropriately. First approach is simpler but aggressive in the sense that both parties will know that the friendhsip situation has changed. Applications like Instagram follow this pattern, an on-off approach. If two users are following each other and one block the

other, both users won't become friends anymore and that means that the blocked user will notice that he has been blocked.

This could be valid on some occasions, but in others, that is not an option. To solve that problem, there's another way to handle that situation which is having a friendship status per user.

If user A requests a friendship to user B and this one accepts, both users will have added to its unique list of contacts the other user and the status of the relationship which means user A will have a relation with user B with accepted status and the other way around. If any of those users, say user B at some point changes his mind, he can do two different options: block the user or delete the user.

The benefit of this approach is that those changes are only affecting one side of the relationship without the other party being aware of that change; to user A, the situation has not changed since the beginning; the only thing that he could possibly realise is that user B will never be online. Obviously, following this schema, user B can change this status back whenever he wants in a safe way.

If what he wants to do is just un-friend the other user, same will happen, user B will never appear connected to user A and more important, user A will not be able to ask for again for friendship.

This is the pattern that will be used on the application.

Obviously, last mandatory use case, is to be able to handle the status of all the current friendships a user has, other way there is no possibility to change them at a given point in time.

3.2.4. Meeting related use cases

The last group of use cases are related to the ability to schedule meetings in advance with a user. This functionality exists to cover the case where a user wants to talk to another but this last one is not available at the moment. There needs to be some kind of way to make the other user know of the intentions of the former.

Those meetings can have a title so it is up to the user who requests it to leave a note.

3.3. Architecting a videoconference web application

At the time of architecting a software solution, there had to be several considerations taken into account. First one, obviously, is that the chosen architecture has to be able to provide a reliable solution for all the problems proposed, the use cases, and another one is to take into account that this solution should provide access to thousands of concurrent users not all

necessarily in a browser environment (there should be room for native handheld devices as well).

Having to develop for multiple types of clients has some implications that affect directly to the architecture. That is, that all the logic regarding the application has to be centralised in the server in order to not repeat that logic in the client.

A scenario where 3 different client platforms would be targeted, for instance, would imply rewriting same code 3 times which is, at least, time consuming (by 3 times) and 3 times error prone as well. The rule of thumb should be to use a server to orchestrate all the application related logic, not videoconference, and delegate to the client the responsibility of representing the data that the server provides.

Another problem that should be taken into account is that every different client platform can be written in its own programming language, which implies that the server has to provide an agnostic way of communicating between the different clients.

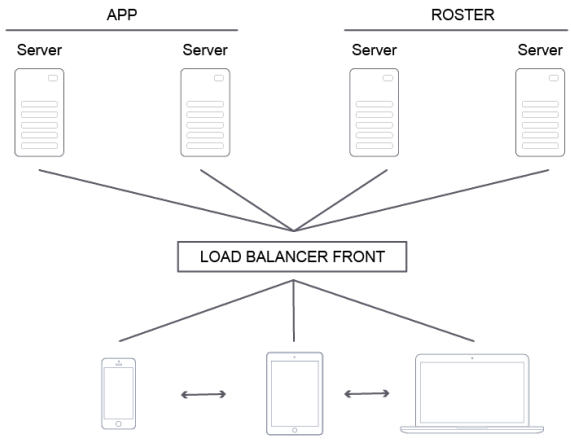

Fig. 3.1 Overall application architecture

Above can be seen an architectural overview diagram. This diagram shows the network architecture rather than the application architecture.

As every aspect regarding software, there are multiple ways to solve a given problem in a proper way, and, regarding architecting a web application, it applies as well. In the following sections there will be a discussion regarding which kind of solution will be applied comparing multiple approaches that could be taken, or if in any case, a trade-off or hybrid between those solutions are needed.

Those decisions are:

- Centralized or decentralized architecture
- Compiled or Single-page dynamic web applications

- Stateless or full-duplex communication protocols
- SQL or NON-SQL database

3.3.1. Centralized or decentralized server architecture

On previous chapters it was stated that the application was going to be a decentralized, peer-to-peer, videoconference application to maximize the number of concurrent connections available and, as can be on Fig. 3.1, there are, in fact, client-server connections. At the end of the day, the proposed architecture is a client-server, peer-to-peer hybrid:

It is client-server oriented because there has to be one place where the app holds the information regarding current state of each user inside the application, and this is responsibility of the main server. All persistence and business logic has been isolated and handled inside that server so, as stated before, there is no duplication of logic across different clients using different technologies.

And obviously, it is peer-to-peer, for all the sensible streaming communications between users.

3.3.2. Dynamic web pages: Compiled or Single-Page applications

Web ecosystem has been always split into 2 types of web pages, static and dynamic web pages. Static web pages are those whose content never changes, and offer exactly the same content to all the users, while dynamic web pages are those whose content will change, showing a custom experience for every user.

Dynamic web pages have been around for quite some time, and there are multiple subtypes of them as well. For instance there are types of dynamic web pages that only rely on making changes at runtime, that is, provided some programmed functions inside the HTML code, those are triggered at certain points obtaining information from servers and updating the status of a given page. There have also been, in the other hand, tailored HTML web pages that are obtained from the server already with the customization code in it as well. That is, they are pre-generated or compiled and then served; an example of that type of web pages is the ones based on JSP (JavaServer Pages).

Depending on which kind of webpage is desired, a different kind of server is needed; each has its trade-offs, essentially, throughput-wise. Seems quite evident that if a server has to provide each client with a different version of a webpage, it’s going to take more time than another who simply has to serve the same content over and over again and that can benefit of caching strategies that could relieve that work. Obviously the more time a server needs to spend in each request, the lesser requests it could attend per second.

In the field of dynamic web pages, there has been a rising trend these last years: single-page web applications. They appeared in order to fix several of the problems that dynamic pages suffer: slow responsiveness and scalability,

tight coupling code and standardization regarding data consumption from the server.

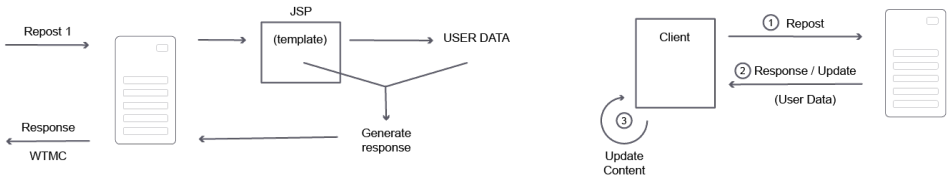

Fig. 3.2 Dynamic web pages: single page vs compiled applications

It is known that HTTP protocol is a request-response protocol, conceived to bring content at each request. The latency between the instant that a client makes a request until the response is being obtained and processed determines the idle time a user will have to wait to see the contents updated (or being redirected to the next web page).

Desktop applications behave in the opposite way; all the client content is already in the application, so if a server is needed, it only needs to get certain state information and this operation is very likely to be happening on the background so the user is presented with proper handling of this idling times with page loaders or the so called, empty states.

Single-page web applications are an attempt to bring those desktop app capabilities to the web. On a first request to the server, the client downloads all the client-side specific content needed to navigate through the different states (a concept analogous to a page) of the web, which will be static. This has a great impact in user experience; there are no waiting times bound to the request-response latency inherent to the http protocol; instead, the client can gracefully switch directly to the next state and be ready when the server data arrives and just show that new data at the very moment it is obtained, without having to wait for new CSS, HTML or Javascript to be loaded and executed.

This targets two of the important benefits of single-page web application, slow responsiveness and scalability: Slow responsiveness in the sense that user now does not have to wait to be redirected between pages because he already has downloaded them previously, and scalability because now that the client code is completely static, it could be cached and that will free the server and make room to support higher traffic loads.

Another of the problems regarding dynamic web pages and web application servers lies in the fact that there is not a clear separation of concerns, which is a hint of clean code and therefore, a cleaner architecture. Both client code and server code are coupled.

In the case of compiled dynamic web applications, there are server variables and methods (tied to the request or the session, for instance) that have to be added inside the HTML client code. That is problematic in multi-disciplinary

teams where pure backend developers and pure front-end developers are forced to work together because they are dependent on each other.

This last problem is solved gracefully thanks to single-page applications, enabling web applications to be treated as a first-citizen client, like mobile clients for instance. They can even be hosted independently from the applications server.

And finally, is also a benefit for backend developers regarding maintainability. Before SPA, or single-page wen applications, they were usually forced to maintain two different endpoints for providing services to the different clients, one, at least, for web applications, that run directly from the same server and have access to the server variables and another one to provide service to external clients, like handheld devices, through, typically, REST or SOAP endpoints.

With this approach, SPA, both type of clients are fed through the same standardized endpoints thus simplifying the number of code to maintain and clearly isolating each member's responsibility.

With all this information, seems quite clear that SPA is the way this project will be implemented.

3.3.3. Stateless or Full-duplex communication protocol

As said previously, there are certain restrictions when building up a system that has to be thought as scalable and ready to work with any kind of client, even those that will be created in the future with technology that currently is non-existent.

At the end of the day, scalability means being able to give service to a higher number of clients at same time, and this, can only be achieved with more servers working all together.

There is only one constraint that must be fulfilled, each server has to be able to provide seamless service to any user regardless of what server had previously answered previous requests that one client could have made.

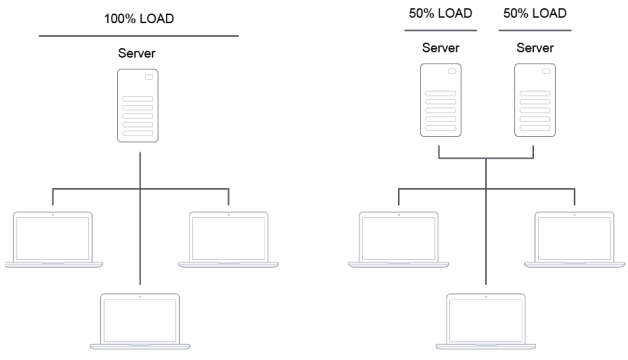

Fig. 3.3 Load distribution between servers

Taking as an example Fig. 3.3, there are two different scenarios explained.

Scenario A consist of an application which is being served by a server and its at its 90% of capacity. That means that the server is about to reach a point of full load and will not be able to provide service to newer clients. That would be the maximum throughput that this example application could achieve.

Now, scenario B is an example of how to try to scale this architecture to reach more concurrent clients; the idea is simple: try to divide the work of one server by adding another one thus creating a server cluster. Theoretically, this means that once this scenario has been set up, each of the servers would be working approximately at 50% of its capacity and neither one of them would not be stressed. Well it turns out, that depending on which type of connection protocol is taken, that will not happen.

Turns out that web servers are designed to work on top of HTTP protocol, which as stated before, is by design stateless, but at the end of the day, plain users of web applications do not care about this and rely only on the scope of a session in a page. We could define a session as the whole interaction process since the user logs into a web page until it closes the session or leaves the page.

Dynamic web applications that have an authentication mechanism just ask for the user credentials one single time and once users are granted the access are not required to login again at each step they do, until they log out.

This session scope at application-level has to be tracked at transport level through the HTTP protocol so at the end of the day, there has to be a way to cheat that statelessness of the protocol and there are two main strategies to do so, a stateless one and stateful one.

If at application level there is a stateful communication protocol, it means that the server that accepts the first request of a given user will have to deal with all the other requests of that user during that session so that the data exchanged keeps being consistent. For example, for J2EE and a Tomcat web server, whenever a new session is created, the server creates a JSESSIONID cookie that keeps track of the user activity. Going back to scenario B on Fig. 3.3, if that user keeps sending each request randomly to both servers, only the server who started the communication will be able to react appropriately since it is the only one who internally keeps track of that session value.

On the other hand, there are the stateless protocols, which are the current trend in the industry. The idea behind them is to provide all the information needed by the server at every request. Recovering the analogy of the user who wants to navigate through an authenticated web application, using this stateless approach, once the user has successfully authenticated, it will be given a user token that it should keep adding (typically on the headers) at each request. With that change, it does not care which server inside the cluster handles each request.

With the information of both modes, and taking a final look at Fig. 3.3, scenario B should only be achieved by the stateless server architecture because with the stateful architecture even though any number of new servers were added to the cluster, the first server would keep having the same 90% load of traffic since all the sessions are bound to it. Only clients starting fresh sessions would benefit of multiple servers.

There is also a final consideration related to handheld devices and real-time applications: due to battery consumption restrictions, it is considered a bad practice for those systems to keep persistent connections with the servers. If no persistent connection, or at least the minimum possible persistent connections had to be used on this platforms then only REST like endpoints are valid connection options but main drawback from them is that connections can only be started from the client-side to the server side.

That means that if real-time event has to be sent to a client it could only be achieved by implementing a polling system where those clients are systematically asking for new data. This obviously affects the scalability of the system drastically so, at this point, a trade-off decision has to be taken.

The final architectural decision is for the second time, a mix of both solutions. On one hand, browser clients can benefit of real-time persistent connections so all the traffic between the app and the server will go through that interface. At the same time, all the non real-time communication traffic on handheld devices will be exposed through a REST API while all the real-time traffic will be held through a persistent connection. That way the best of both worlds is taken into consideration in the platform while minimizing the potential high-battery consumption on handheld devices.

3.3.4. Data persistence

After talking about scalability issues in the previous section there is only on topic to decide on architecture-wise: data persistence.

It is obvious that if multiple servers can be dynamically spawn and shut down according to traffic loads and that each client could potentially reach any server for any given request, the data that keeps the state of the application has to be kept in another isolated environment that all the servers could reach in a reliable way, that means that regardless of what server is reached, the information has to be consistent.

In the case of this application, since data is stored in a structured way, the only way to achieve that is through a database server. Turns out that database are split into two different types of them: relational and non-relational databases.

Relational databases organize data into one or more tables or relations of columns and rows, with a unique key identifying each row. Rows are also called records or tuples. Generally, each table/relation represents one entity type (for instance a customer or a product). The rows represent instances of that type of entity and the columns represent values attributed to that instance

and typically most of them use SQL, structured query language, as the query language to obtain information for those tables.

On the other hand, non-relational databases, as the name indicates, does not store the data in a different tables related between each other. There are a lot of different kind of this so called non-relational databases according on how do they manage the data, some of the examples are:

- Key-value store
- Document store
- Graph

Key-value database use a data storage paradigm designed for storing, retrieving, and managing associative arrays, a data structure more commonly known today as a dictionary or hash. Dictionaries contain a collection of objects, or records, which in turn have many different fields within them, each containing data. These records are stored and retrieved using a key that uniquely identifies the record, and is used to quickly find the data within the database.

Document databases store all information for a given object in a single instance in the database, and every stored object can be different from every other. This makes mapping objects into the database a simple task, normally eliminating anything similar to an object-relational mapping because the object stored is directly the object that the application is going to use so there is no mapping needed.

Graph databases use graph structures for semantic queries with nodes, edges and properties to represent and store data. A key concept of the system is the graph (or edge or relationship), which directly relates data items in the store. The relationships allow data in the store to be linked together directly, and in many cases retrieved with a single operation.

At the end of the day, the main idea behind the non-relational databases and the reason why they are getting popular is because while relational databases forces the data modelling of the entities of any given project in a way that can be structured in tables, non-relational databases don’t add constraints of that kind, its just a matter of picking the one that fits better the application’s needs.

In the context of the application that is being architected, in order to choose which database is going to be used, it is mandatory to know which are going to be the entities to be persisted. Fig. 3.4 shows the entities needed according to the requirements.

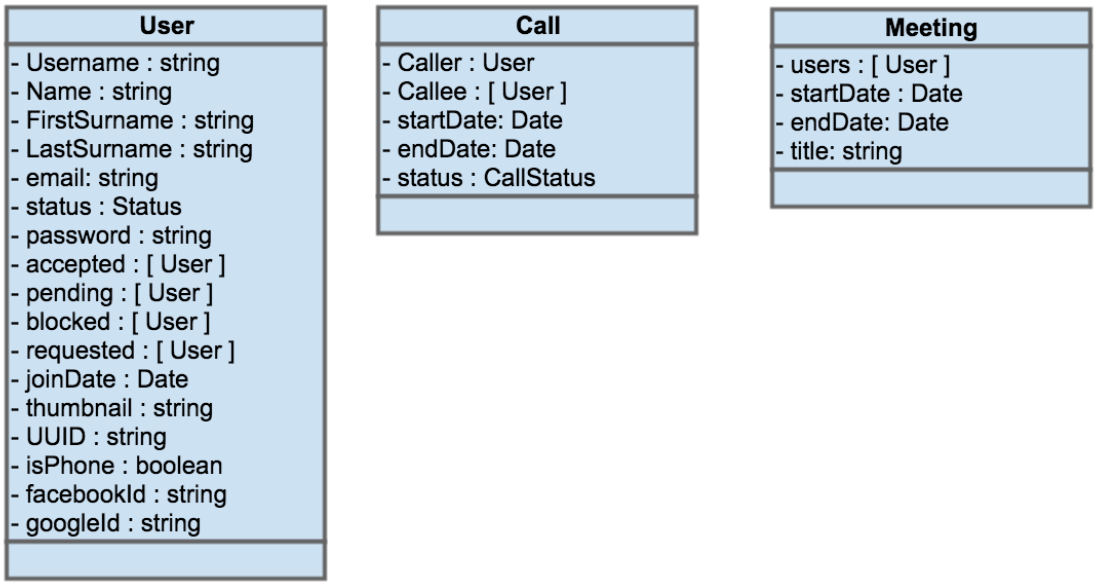

Fig. 3.4 Entity diagram

As can be seen, there are 3 entities: User, Call and Meeting. They all rely on user so there is a kind of relation between them so it could be modelled like a relational database. Being said that, depending on how data wants to be presented to the user, could be also easily implemented by a document-based non-relational database.

For instance, the dependencies are unidirectional, between Call and User Meeting and User entities, and means that they only need user data to represent that user in the screen and not all the User entity fields needed to do such thing. Knowing that, a technique called entity embedding could be used to take advantage of that: instead of creating a relationship between both entities, a subset of the User entity could be embedded inside Call and Meeting databases containing only what is needed, in this case, the id, full name and image data.

Doing that change, an increase on speed at query time is obtained since the server does not need to realize JOIN queries to obtain data each time there is a need to show a Call or a Meeting thus getting a more responsive application.

With all this being said, this is the final aspect of the application architecture, on the next chapter the implementation details of every aspect will be explained.

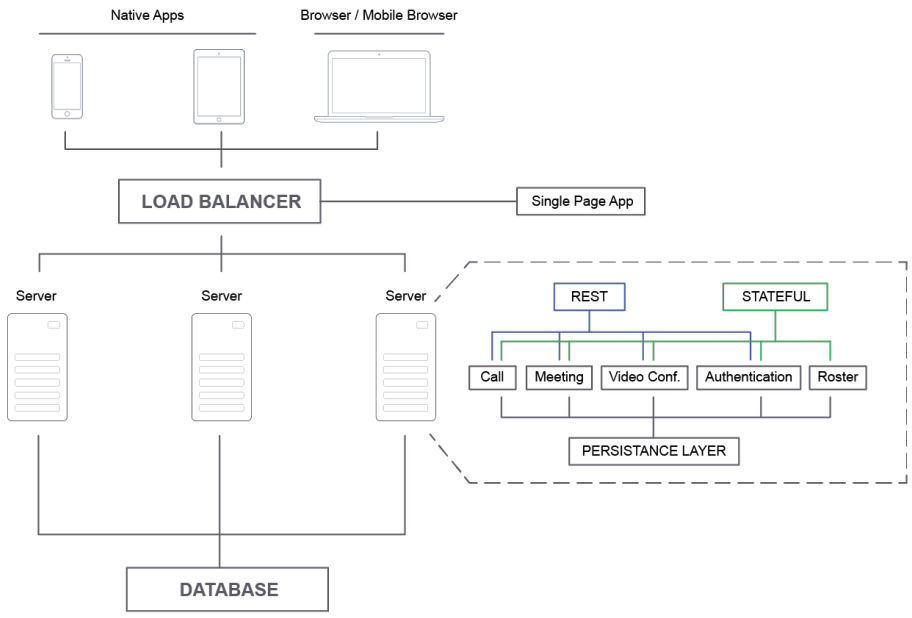

Fig. 3.5 Final architecture design

On the final figure, it can be seen how all the topics that have been discusses through the whole chapter fit in. On one hand, there can be seen how this architecture can be horizontally scaled through adding more servers and how the internals of the server are providing a user all the information needed to fulfil the specified use cases.

It can be seen as well that each server will expose REST endpoints along with a persistent connection to interact with that will be the bridge between the clients and the server.

All of this will be orchestrated through a load balancer which will handle gracefully which server should address each request as well as serving the static content that holds the Single-Page application web client.

Finally, all the data will be stored in a non-relational and document oriented database to ensure the quickest and easiest of the interactions with the user per request.

On the next chapter, all those schematic boxes will be put in context and explained in detail the implementation that converts this architecture from theoretical, to a real app.

CHAPTER 4. VIV, the multimedia and multi-conference web application

In this chapter, implementation details that conform the application that has been developed for this project will be given. It will start exactly where the previous chapter finished and explain which are the technologies that make possible the theoretical concepts gathered through previous chapters.

The content will be split across both server and client components and will talk about the technologies used, and how they blend together to create the final application.

4.1. Server components

4.1.1. The MEAN stack

Traditionally, Javascript has been always a language fully oriented to client technologies, specifically for web pages. As stated before, whenever a page had some animation or some kind of effect out of the usual navigation flow jumping from hyperlink to the next one, there was quite possibly, Javascript handling all those effects.

Although initially it had a bad reputation amongst developers for their lack of proper tooling to work with, this trend has changed drastically and now the current trend in the IT sector seems to be Javascript focused. From backend development to frontend without forgetting the real players nowadays: handheld devices; even native applications can be developed for multiple mobile clients with Javascript's help.

On the web side, it seems reasonable that Javascript has won some space between the usual suspects as a backend language. At the end of the day, the big win would be to be able to write from top to bottom an application with one single language and that's been the aim of this project's application. LAMP (Linux, Apache, Mysql, Php) stack has a new contender; the MEAN stack (Mongo, Express, Angular and Node).

Fig. 4.1 Mean stack and its components

Node is simply a Javascript compiler based on the V8 Javascript engine developed in C++ and used on chromium based browser (the open sourced engine of Chrome).

Actually it does more than compile the code; it actually can execute it thus allowing developing programs written on Javascript outside of the browser environment.

Developing on Javascript has quite benefits one of which is specifically targeted to MVP projects, quickness. The amount of needed code to be written on Javascript to achieve some tasks is considerably lower than in other languages, like for instance Java and on top of that, if done properly, code can be as performing –if not more- than Java itself.

Node sports NPM, stands for Node Package Manager, which is a dependencies manager, which would be the somewhat equivalent to maven for Java-based projects or Maven.

With those two components, NodeJS becomes a reliable language to be used on professional environments as well because it allows creating libraries that could be re-used and shared across the web, speeding tremendously the development of any task.

On top of that, the fact that NodeJS is a virtual machine means that the code can be executed in any environment as long as a virtual machine for that given environment exists.

Next letter from the MEAN acronym is E, from ExpressJS. ExpressJS is a web framework designed to work inside a NodeJS environment, so it actually is just a library dependency that enhances a NodeJS application with web server capabilities.

For instance, comparing the amount of code needed to deploy a hello world REST API between those two languages, using Express in Node and Spring MVC on Java, while on the Java part there are needed a J2EE web application server, the xml definitions and the java classes that need to be compiled into a war file and deployed in the web server, say, Tomcat, with NodeJS below is a complete server application:

```
var express = require('express')
var app = express()

app.get('/', function (req, res) {
  res.send('Hello World!')
})
```

That's all what is needed to run a server from a NodeJS application with ExpressJS. To execute it suffices to save this snippet in a file and execute it through the command line:

```
$node filename
```

ExpressJS layer will take care of the REST endpoints that will provide service to the handheld devices using a native app.

So far, it has been covered half of the MEAN stack acronym. Next Acronym's letter, A, stands for AngularJS, but this one will be commented on the client section of this chapter, so then, the last letter that needs to be presented is M that stands for MongoDB.

MongoDB is the selected non-relational database for the project. As commented in the previous chapter, it is a document-based database, which means that stores all the contents on a de-normalized fashion. The whole point of selecting this kind of databases is simply the fact that it suits best the use cases that compose the whole application and in this case, that's also true.

Obviously, MongoDB runs outside of node's environment because it has its own server (according to the general architecture graph on Fig. 4.2), so there needs to be a bridge between Node's server and MongoDB, that's Mongoose.

Mongoose, like ExpressJS, it's a Javascript library; more accurately, it is an ODM, an Object-Domain Mapper, that is able to map Javascript objects to MongoDB documents. Also it provides the methods to query, create, update or delete those documents.

MongoDB stores all the data in collections, each collection representing one entity. On top of that, the fact that there is not a table scheme concept like on relational databases means that an instance of a persisted object of the same collection might not have the same fields exactly as another instance of the same collection. That's because the idea that holds this kind of structures is to keep only the data needed for the client and that keeps reinforced by the fact that the database stores the information on JSON objects.

So everything makes sense on the backend side; from top to bottom all the used tools are aligned: The business code of the backend is written on Javascript, exposed for external consumption via REST API also through ExpressJS web server framework and data storage on the same format that will be consumed, JSON, on a MongoDB database and all running inside Chrome V8's virtual machine, NodeJS.

The following figure shows this in a more structured way

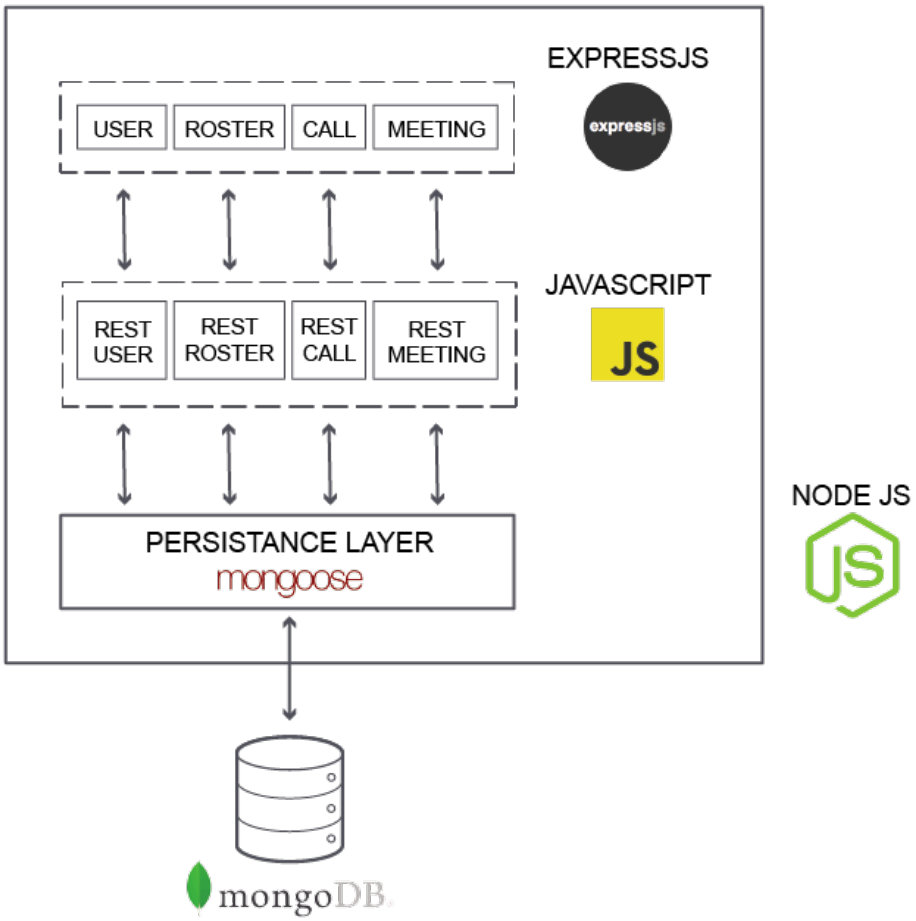

Fig. 4.2 MEAN stack on VIV application

To complete the backend functionalities, there is another interface that needs to be exposed for the clients to consume data. A full-duplex persistent connection. To achieve that, the MEAN stack will need to be extended.

4.1.2. Extending the MEAN stack: adding websocket connection

REST interfaces are actually the common way to expose any service for its consumption regardless of which kind of client is consuming it. It's cheap, quick and quite performing but has a big problem when dealing with real-time applications: connections can only be initiated from the client.

In the context of a videoconference, there are, at least, two users with different roles that need to cooperate in a short window of time: the caller, with an active role, and the callee with a passive one. In this scenario in a REST environment, the callee simply can't be reached by the server and thus, the videoconference will never happen.

There are several solutions to resolve that problem in a REST-only environment, but none of them are optimal. On one side there are polling

techniques and on another side, if dealing with handheld devices, there are push notifications.

Polling is simply, querying continuously the server so the moment the new information is available, it could be obtained. That obviously comes as a trade-off for scalability issues. Imagine a polling rate of 250 milliseconds with 500 users simultaneously. That adds up to 4 polling request per second per user, which adds up to 1000 request per second. That simply doesn't scale.

There are 2 alternatives to this: First one are the SSE, server sent events (which means, keep a connection-opened with the server so whenever there is a new event that involves a given user, the server could rely on this connection and send an event through that). It would be a half-duplex solution since this connection can't be used for the client to inform of real-time events the server on the other way around. The second ones are Websockets.

Websockets are simply put, TCP sockets that work on the browser. It is a persistent, full-duplex and low-latency solution to communicate both ends. Unfortunately, not all the clients at this point support them so if there is not a reliable way to take advantage of them it cannot became a real solution.

Fortunately, another NodeJS framework comes to the rescue: Socket.io.

Socket.io is an event driven communication protocol. What makes socket.io special is that even though it is designed to be used over websockets, it can gracefully change the transport protocol used underlying according at the browser specifications, from preferred way to worst case scenario. That means that per instance, socket.io could have a configuration where it is stated that thee list of connectivity strategies are ordered: websockets, long polling, polling or error if none of that works.

The important piece is that this will work transparently for the user, which will have the same experience regardless of how what protocol is being used.

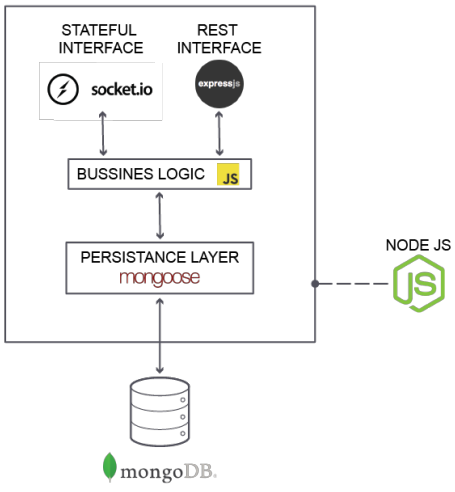

Fig. 4.3 Full backend architecture for VIV application

As explained before, websockets main purpose is to provide a bidirectional way of interacting with the other end (the server in this case) from the browser and on top of that, it has the benefit of being a raw TCP connection, so there is no overhead due to HTTP headers being sent.

On the other hand, that is a drawback per se, because it means that its completely up to the developer to create a control logic to be able to issue different types of information, otherwise, there would not be a way to distinguish the kind of data being sent through it.

That's where Socket.IO is handy. It is designed as an event-driven data channel, which allows sending different events with its related data thus making quite easy the task to identify what is being sent.

Another powerful feature that Socket.IO has are the rooms concept. Imagine three Websocket connections from three different clients. This will be expanded later on this chapter on the roster implementation details.

4.2. Client components

4.2.1. AngularJS

AngularJS is the last letter in the MEAN stack acronym that needs to be explained. It is quite different than the other parts of the stack; while all the other parts relate to the server, this part deals with the client side of the application, that is, what the user will interact with.

As explained before, the fact that using AngularJS (or any other framework that allows the developer to completely separate the client logic from the server one) is one of the main reasons why it has become a big player in the industry, but it is not the only one.

There are several great improvements that AngularJS brings to the web development that justify its big adoption rate; there are enforcement of design patterns like MVC, MVVM, Single responsibility or observer pattern (applied in the star feature double-binding), dependency injection enforcement and testability.

To sum up, it makes writing client side code in a maintainable and robust way:

MVC (Model-View-Controller) and MVVM (Model-View-ViewModel) patterns, enforce the single responsibility law, each part is only responsible of one thing, for instance, the model holds the information that wants to be dealt with, the view holds responsibility of how this information should be rendered and the controller is the one who holds the logic to adapt the data from the model to make it suitable for the view.

On the other side MVVM implements the observer pattern, where the view registers itself to receive any kind of change in the model object so it could be updated instantly. Greatest benefit of this pattern is that it goes bi-directionality, that's why it is called double-binding, because it could be as well that the view updates the model as well, and everything would be synchronised. For instance, typical example scenario to understand that are the HTML forms. Once the form is presented to the user, the model will have the previous information of the user, so it would be pre-filled (that's the ViewModel updating the view), the moment the user updates one of the fields, it is being updated the other way around, the View interacting with the ViewModel. As can be imagined, having the state of objects synchronised directly while preserving separation of concerns principle is a huge benefit that not only makes code better and more understandable but also enforces testability.

Finally, last topic, dependency injection. That's arguably the best feature that this framework provides; The AngularJS injector subsystem is in charge of creating components, resolving their dependencies, and providing them to other components as requested which means that there is no need to share instances of objects on shared scopes and on top of that, makes components modular. That means simply put, that as a developer, components –that are well architectured because the framework enforces to do so- can be created modularized, and externalized as a library, which means that every application can benefit from them.

4.2.2. Architecting the client-side browser app

The fact that AngularJS enforces and benefits at same time of building modular application has a counter effect, what if a legacy application relies on several external modules that after a while have been updated with new features and deprecated features that where relevant to the application? It would end up breaking the application and not being usable anymore. Other systems have solved that problem several years ago with dependency management software, as mentioned before, Java has maven for instance.

The point is that, this could be a real problem for mid and long-term applications and therefore, a solution is needed. In other words, AngularJS has to rely on other external tools that solve this kind of problems as other technologies do. To solve this and other similar problems, there are tools that helps setting up the environment for this Single-Page applications to work; in this section a brief description of all of them will be made:

The dependency management is done by BowerJS. BowerJS is a NodeJS utility that simply keeps track of each external module needed to successfully build the AngularJS application as well as the version being used. That way, as Maven, it will keep track of all the dependencies thus fixing the previously described problem.

GruntJS is a Javascript task runner. It deals with automation and orchestration of tasks for the client application. For instance, it can create a local server to

test the client directly without the need to deploy it to any place, the execution of unit tests, minifying (obfuscating the code as well as merging all the Javascript different files into a single one to increment performance) and all this kind of tasks that are needed for a healthy application development environment.

Finally, the last important tool used is Yeoman. Yeoman is a scaffolding framework designed for AngularJS based applications. Simply put, is tool that according to user preferences creates a base application with all the resources needed. Essentially, if GruntJS mission is to keep a healthy environment, Yeoman creates the environment.

It allows the user to select which AngularJS version to start from (some of them are quite different from the others, like branch 1 to 1.5.x from 2), which SCSS to use, if any, in this case Bootstrap 3 as well as generate the initial bower dependencies file, grunt configuration file and ready to work hello world app, all of this within a minute, really convenient.

With the help of all those technologies, the client side application can be easy and enjoyable to work with, maintainable and robust, because with a simple check after downloading it from the repository can be built and verified.

4.3. Analysing end-to-end components

Roster component

Roster component is one of the key components in the application. It is needed in order to search users and also displays an indicator showing the current state of a user, if it is connected, disconnected or on a handheld device.

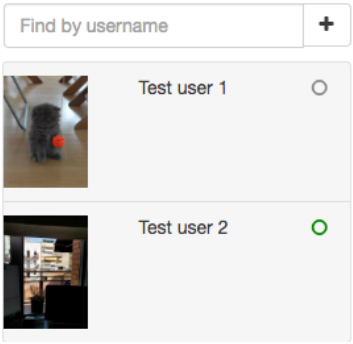

Fig. 4.4 Roster component view

This is one of the components that need a persistent connection because it is mandatory to track as accurate as possible the status of a user so that it could be called, otherwise, the UX, user experience would not be satisfactory.

To do so, as explained earlier, this component will need to interact with the Socket.IO interface from the server.
This component needs two different sets of information; on one side, the list of users that have a relation with the logged in client: username, image and full name. On the other hand, it will need the connection status.

To illustrate how this works, consider a scenario where there are 3 clients, A, B and C. To simplify things, consider as well that all of them are already friends with each other and that B and C had already been logged in the application. This whole example starts from the moment where A logs in and has to build his roster; Fig. 4.5 on the next page show schematically the process about to be described.

This task, that could be really complicated gets quite simple thanks to Socket.IO's built in functionalities. When client A gets on the landing page of the application, it triggers a websocket call to retrieve his list of contacts. When that event reaches the server, this one does a set of actions: first queries the users from the database and right after that sends back that information through client A's websocket.

At this point, client A will have a list of all his contacts but won't have any information about the state of each particular user so by default all of them will be shown as disconnected.

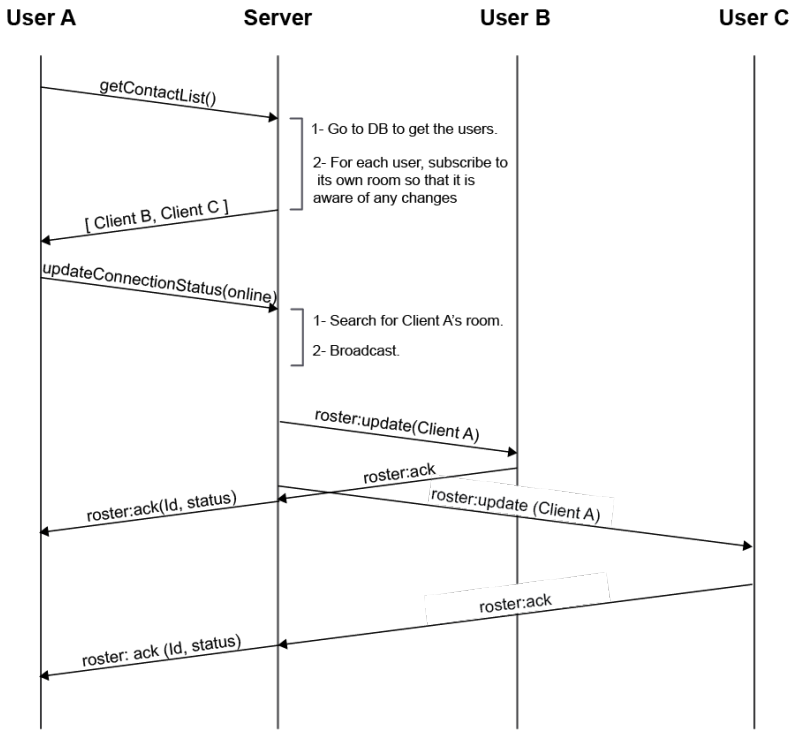

Fig. 4.5 Roster update logic

On the other hand, the server after sending client A's list of contacts still has a couple of things to do: first, registers the socket from client A to client B and C's rooms; from that point onwards client A will be aware of any event broadcasted on any of those rooms. Finally once the server has registered client A to all his contact rooms, it broadcasts a message to client A's room announcing a roster:update event with the status of the new connection user.

Since client B and C were connected prior to A's connection and being contacts of A, they were previously registered into room A (followed this exact process before) so they will receive that broadcasted event from the server and therefore, update client A status to reflect that now is online.

Last step happens on the client, once they process the roster:update event, in this case, B and C will answer back sending a roster:ack event message informing of its own status that will be delivered back to client A.

Finally client A will receive a roster:ack message from each of the clients thus will update the status of the connection for each of them. At this point all the clients have an updated and synchronised version of the status of their rosters.

Handle this without Socket.IO functionalities would be way much harder than having 3 different events. Same benefits are used for the signalling channel when a user wants to initialise a videoconference with another one.

Videoconference component

This is the other component that is going to be explained because it's the core functionality of the application. The biggest issue is to create a communication protocol between a client, user A, the server and another client, user B that on the other hand could be extensible to multiple users in order to achieve a multi-videoconference.

The problem can be split in two halves. First half is dealing with the request of the call; an analogy in standard non-VOIP calls would be the dialling and ringing state. The other half is the signalling process inside the conference page.

First part involves, conceptually, two steps; First step is when user A at any given point, calls user B. From a UX point of view, the sequence should be as follows: User A selects User B from his roster and presses the call button. That opens a confirmation dialog that if confirmed, initializes de call with User B. At this point, User B displays a call dialog screen prompting him to accept or deny the call. If no answer is provided, in order to avoid lockdown states in both ends, a countdown of 15 seconds is added as a ringing limit. If User B accepts the call is brought to the conference page where he will wait for User A. User B is on the second step. Meanwhile, User A is given the confirmation back from User B so he enters as well at the conference room.

To achieve this use case Socket.IO, will be used as the previous use case. Below can be found the call request flow explained:

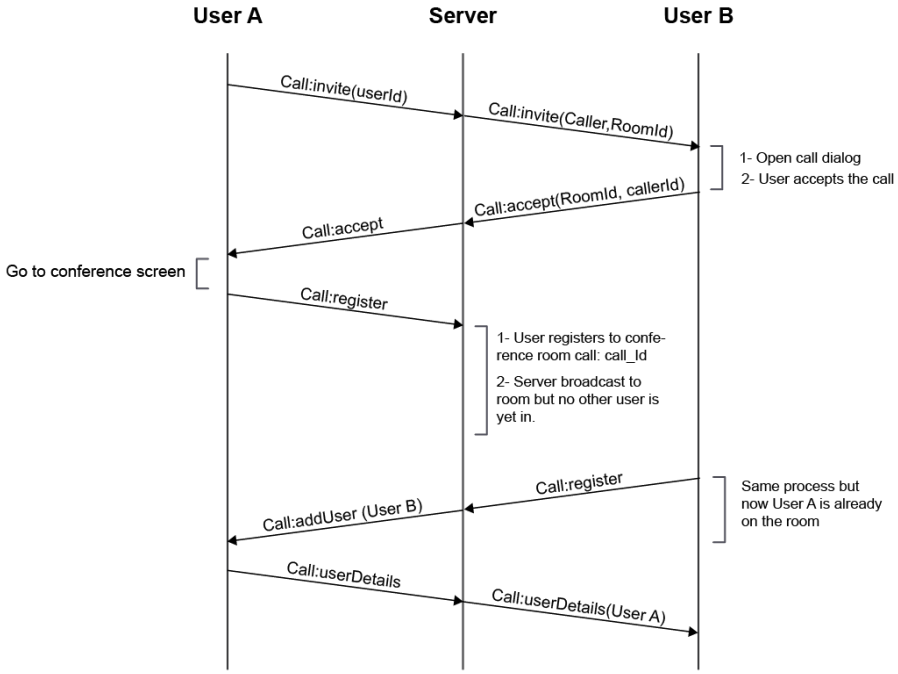

Fig. 4.6 Call request flow

The UX flow explained above is summed up on the first round-time trip composed by the call:invite method sent by user A and the call:accept sent by user B. Visually, once user B receives the call:invite event, shows a popup with caller information (username and picture, so that he would be able to identify who is calling). Upon acceptance, user B goes directly to the conference page and at same time, sends back the call:accept event to user A who does the same. At this point both users are in the conference room and are ready to start the create offer/send answer WebRTC protocol through but none of them has information about which role has to play and, as said before, this can't be limited to 2 users, so if more users later want to join, it should be possible.

As explained before, main problem here is the race condition that incurs on who has to start the process and who needs to passively wait without doing any action in a way that if later another user arrives, it would perform the same way.

The race condition happens because first thing the user has to do once it has landed in conference page is request permission for the local streams through the GetUserMedia API. It is not as simple as stating that the first user who reaches the page would be the initiator (in this case User B because he accepts the call and once he does that he's brought to that page while User A still has to wait for the event emitted by user B to propagate through the network, receive it and process it) but who is the first who grants access to the resources.

Once the webcam and or audio devices has been requested and granted, then the process starts. It is the same process for both parties so this way it could be repeated no matter how many times or when a new user could reach the conference and goes as follows:

First the user after having their local resources allocated sends a call:register with his id and the room's id (generated by the server when the call was created during the call:invite process). Once this event arrives to the server, the user is registered in the virtual conference room created by that call and at that point, server broadcast at that conference room a slightly different message, call:addUser with user details.

Main benefit this message broadcasting provides is that this will fix the race conditions problem because If User A and User B where to send this message at same time, since the server is single threaded, request should have to be queued thus at the time the first user would be registered, say user A for example, there wouldn't be any other client registered in the virtual conference room so even though the server throws a broadcast message, it simply won't have any socket to send it through. It is only after the second user, user B is registered, that the message call:addUser would be broadcasted to the user A. The biggest problem is fixed and on top of that, this scales to any number of users so the multi-conference challenge will be solved as well!

After that, the events are quite simple, whomever user receive the call:addUser with the information of the other client, will have to answer with its details with a call:userDetails message so that both ends have full information regarding each other. Those messages will be broadcasted so that if there would be, say 3 users in a conference, same message would be delivered to all parties, again, except the sender. At this point each user knows how many participants are in the call: as much as call:addUser plus call:userDetails are thus each user knows how many resources need to allocate (number of <video> html tags needed to be rendered in the conference, one per participant plus the local stream).

There is one last issue that needs to be taken care of. The fact that the call could be for more than a user, and that each user has to keep track of each user's offer/answer flow (a multi-conference is at the end of the day a set of one to one conferences between all the elements of the call, so if there is a 3 people multi-conference, each user will make 2 videoconference, one with each of the other users and will all be rendered at same time).

The easiest way to solve this is send inside the room a direct message just for one client, not broadcasted like the others. This way, server will have to handle the right delivery according to the message he receives. Taking a look at the graph below will help understanding it:

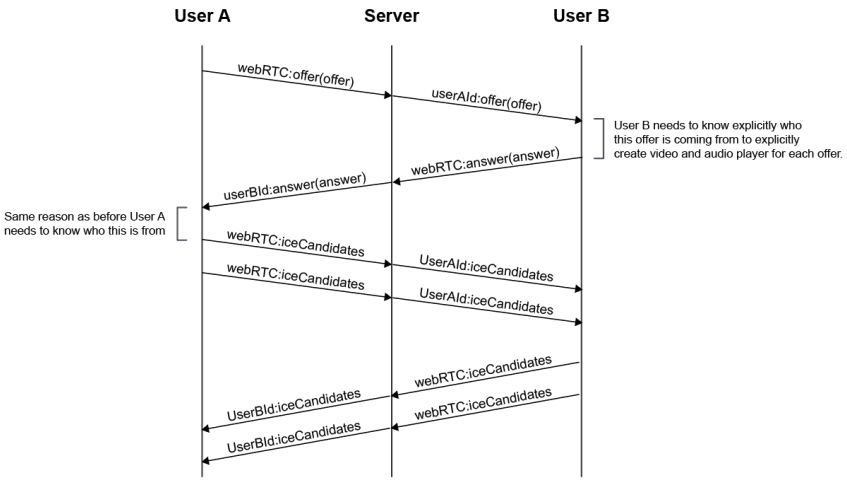

Fig. 4.7 Signalling protocol for offer/answer exchange

User A sends creates an offer and sends a webrtc:offer message with the offer and the user B's id. With that information, when the message reaches the server this one processes it and delivers to User B the content of the offer User A has sent him but with a new dynamic message type composed by UserAId:offer where UserAId is, as the name suggests the id of user A.

This same process will be used for the answers and the iceCanditate exchanged (this is the IP and Port discovery process, so that both User A and User B can speak directly).

Once all those messages have been exchanged, the videoconference can be started.

After this chapter, on the conclusion the results will be shown and explained and afterwards compared with the initial goals to see if they have been achieved or not.

CHAPTER 5. Conclusions

5.1. Objectives achieved

After having talked about all the steps involving the creation of a videoconference app, from its internal components and how they work to the implementation details as well as the needed use cases, it is time to talk about the goals that were set before the project started. In this section will review each of the objectives and see if they have been successfully achieved.

In this case, the main objectives where two: first one studying if with WebRTC is possible to create a videoconference application and secondly, if that first point was proved feasible, try to see if it covers as many corners and use cases as to consider extending it until becoming a white-labelled application that could replace the actual rulers on the sector, Whatsapp, Google Hangouts, Skype or Facebook messenger.

First goal was split on two big subtasks in order to achieve it; the first two chapters were needed to gather knowledge about how the videoconference module works internally as well as how content negotiation had to be set in order to be able to do a one on one videoconference. After that, the second part of the app showed how to implement it without losing the focus on scalability, not only on infrastructure but also inside the application thanks to a custom communication protocol able to deal with an undefined number of users per videoconference dynamically.

Regarding the first goal, not only it has been proved to be theoretically doable but also actually it has become a real application, Viv, which is capable of videoconferencing or audio calling, is even ready to share desktop between parties.

Second challenge implied a deep scrutiny over the top-selling apps available in the market and their key features. WhatsApp, Skype, Google Hangouts or Facebook Messenger have as a core component the text messaging solution and as an addition, they all provide voice and video messaging only as an extra feature.

In my opinion, the answer to the question is yes. Obviously this application is only a prototype but it targets the most complex use cases those applications could have which is audio and video. The other functionalities that they offer are not multimedia but social. On the other hand, there is one thing that would need to be implemented to be as effective as Skype: an MCU. Skype offers an MCU endpoint when there is no way to connect two different clients through peer-to-peer discovery protocols –mostly due to both users being behind NAT environments with all ports closed-. In that case, this application would fail to connect while Skype could achieve that by putting one of their servers as the central node of that videoconference with all the money waste that this implies.

Rather than that, all the other features are on par to what it has been proposed on the application.

5.2. Future improvements

This application project has been under development for 3 years. During this period of time several things have changed, some relating with WebRTC and some other related with the development trends and IT changes.

Relating WebRTC there has been some downsides because the technology was simply not mature enough. At the early stages, the API was designed to allow for recording streams. That would be a really convenient feature to have added in the project since there is a high chance of the user who has to be called not being on the application at the moment and this way, the answerphone concept could me also mimic from the traditional phones.

Another update on the same direction due to the fact of being in a web application environment would be to implement push notifications for web applications. This would remove the previous problem and make any user instantly aware of someone wanting to speak to him.

Regarding bandwidth optimisation, there are a few improvements needed to be done, one of the is handling the addition of a MCU unit when a videoconference reaches a maximum number of users so that each participant doesn't have to be sending the same local stream simultaneously to the other ones. For instance, a threshold could be set where when a videoconference reaches 5 participants it switches from a multi-conference to a simple conference between each user and a server acting as a client or as said before, when ICE protocol fails to obtained open ports for both clients and therefore no connection can be made.

Thinking about aesthetics, it was not one of the main points of the application so there has not been a big investment of time on that matter. To try to fix that, Google has come out with a set of design rules and components called material design that seems like a perfect match for the project.

Finally the biggest step ahead that could be done for the application is adding support for native iOS applications. This way the whole spectrum of the mobile ecosystem would be covered but on the other hand, this would take, at least, another master thesis.

5.3. Environmental study

This section is always tricky with software-based products or solutions. In this case, there is not any advantage using WebRTC rather than using any other technology used by the competitors since, at the end of the day, needs the same kind of components to run, being servers running 24/7 the worst part of it.

On the other hand, it is true that due to its decentralised architecture, the server could achieve a higher number of concurrent connections, which means that fewer servers will be needed to give service to, theoretically, the same number of clients than any other solutions. That definitely would take a positive impact on the environment.

Another thing to think about is that if applying a general rule where the fewer the resources needed the lesser the environment impact of some new technology, it should be considered beneficial as well the fact that developing a solution for multiple platforms would avoid the need of having multiple development teams, each for every single platform. Considering Android, iOS, Web and backend. That means that at least, this project needed twice (not considering Android since a native client has been developed) less resources than any other competitor.

5.4. Personal conclusions

Being able to complete this master thesis has been something really enriching on a personal level. This project started as a self-challenge given my job at the time of starting it. It was a real necessity that we needed to cover and the technology was completely brand new so there was a chance of merging my personal interest and the work I did by then for a living. It proved worth it and this technology is being used right now on it.

This project has given me the opportunity to compliment my studies with extra-curricular content to the master thesis lectures like for being able study from its conception how has a technology that has changed how multimedia contents are delivered today been evolving from only drafts to real implementation and how the open source community works together to achieve a common goal which everyone could benefit.

On a more personal insight, when this project started 4 years ago, gave me the opportunity to choose a topic that motivated me and that definitely has been really useful for my professional career to the point where it ended up just being a master thesis proof of concept to become a pet project I worked on weekends and spent time trying to improve. It made me learn new technologies like WebRTC itself, a whole new development paradigm with Javascript and NodeJS for backend, AngularJS as a front-end web client and created a native Android application using NDK and the compiled libraries from WebRTC's repository.

This has definitely had an impact in my professional career to the point of getting job offers related to WebRTC technology.

As a conclusion, I couldn't be more satisfied to see what I've been able to achieve thanks to all the knowledge gathered through the master.

BIBLIOGRAPHY AND REFERENCES

BIBLIOGRAPHY

Johnston, A., Burnett, D., *WebRTC: APIs and RTCWEB Protocols of the HTML5 Real-Time Web,* Digital Codex LLC, New York, March 2014.

Web Real-Time Communications Working Group (initial version), URL: <https://www.w3.org/2011/04/webrtc/>

Web Real-Time Communications Working Group (updated version), URL: <https://www.w3.org/2015/07/webrtc-charter.html>

WebRTC official site, URL: <https://webrtc.org/start/>

Browser support scorecard, URL:<http://iswebrtcreadyyet.com>

SDP for the WebRTC draft, URL: <https://tools.ietf.org/html/draft-nandakumar-rtcweb-sdp-08#section-1>

Javascript Session Establishment Protocol draft, URL: <https://tools.ietf.org/html/draft-ietf-rtcweb-jsep-08#section-3.1>

Escola d'Enginyeria de Telecomunicació i Aeroespacial de Castelldefels
UNIVERSITAT POLITÈCNICA DE CATALUNYA

ANNEX

TITLE: Study, design and implementation of WebRTC for a real-time multimedia messaging application

MASTER DEGREE: Master in Science in Telecommunication Engineering & Management

AUTHOR: Xavier Lagunas Calpe

DIRECTOR: Juan López Rubio

DATE: February 17th, 2017

ANNEX A. FULL APPLICATION USE CASE SCENARIO

This appendix will show some of the common use cases mentioned in the document. To do so, it will be described the whole process since the user registers in the application until the user starts a multi-videoconference.

To begin with, the user types on the browser the URL of the domain that points to the server where the load balancer, and NGIX server, is hosted. Which leads him to the login screen.

Fig. A.1 Login screen

Once the user lands there, he realises he has to register himself on the application to be able to use it so he clicks on the register now link.

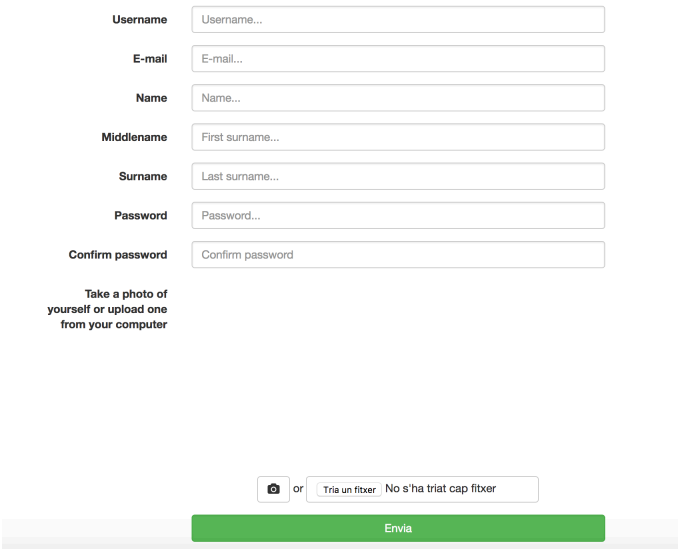

Fig. A.2 Register form

Once the user successfully fills in the form and thus gets registered, is automatically redirected to the landing screen.

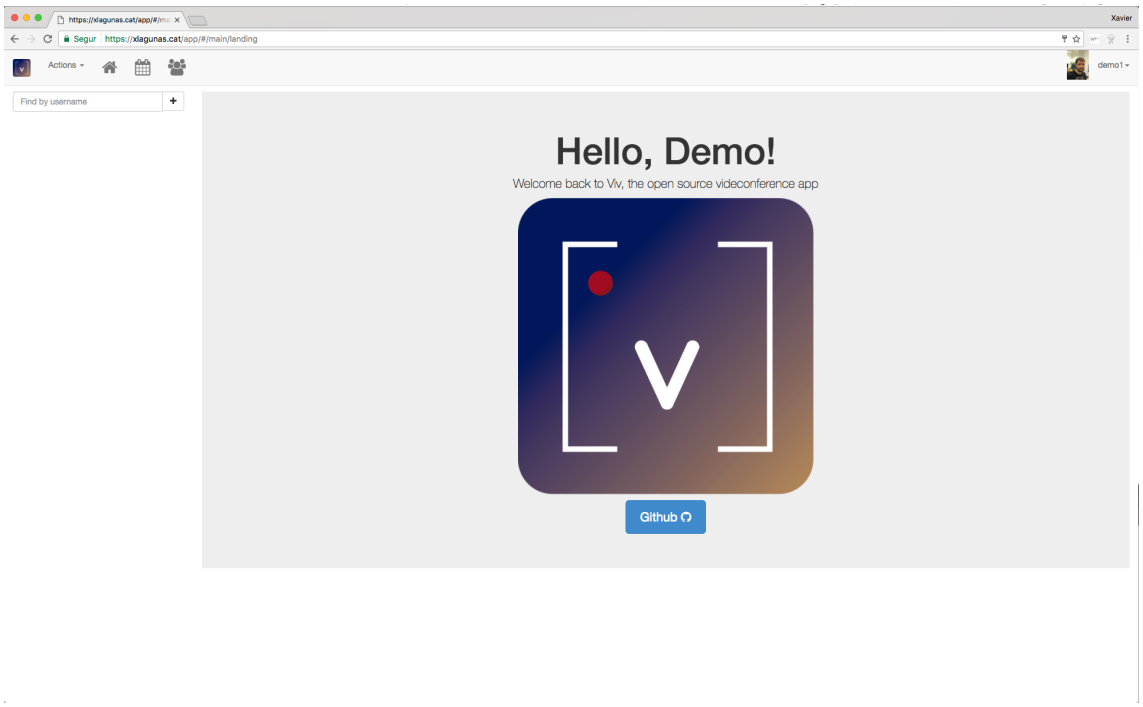

Fig. A.3 Landing screen

Now he proceeds searching for his friend on the search bar in order to send him a friendship request, so he types his name and clicks the search (+ sign) button.

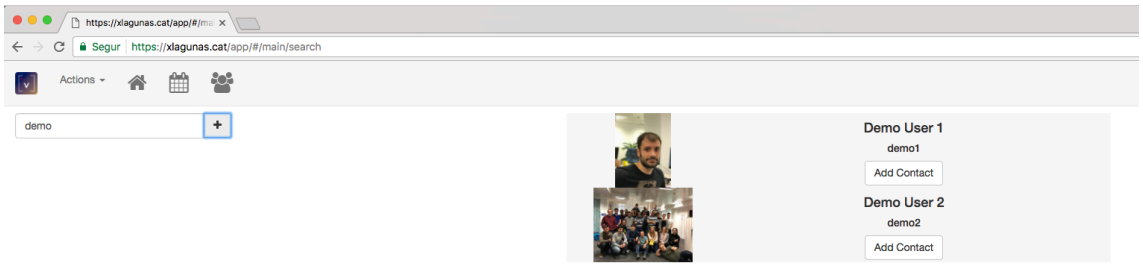

Fig. A.4 Search view showing users matching the criteria

He finally finds his friend, Demo User 2, so he adds him to his roster contact. At this point, the request is on a pending state until Demo User 2 authorizes it himself.

Fig. A.5 Real-time contact request notifications

On the other end, Demo User 2 notices that some friend related change has happened just checking how, instantly, the contacts icon turns orange so he clicks on it to see the changes.

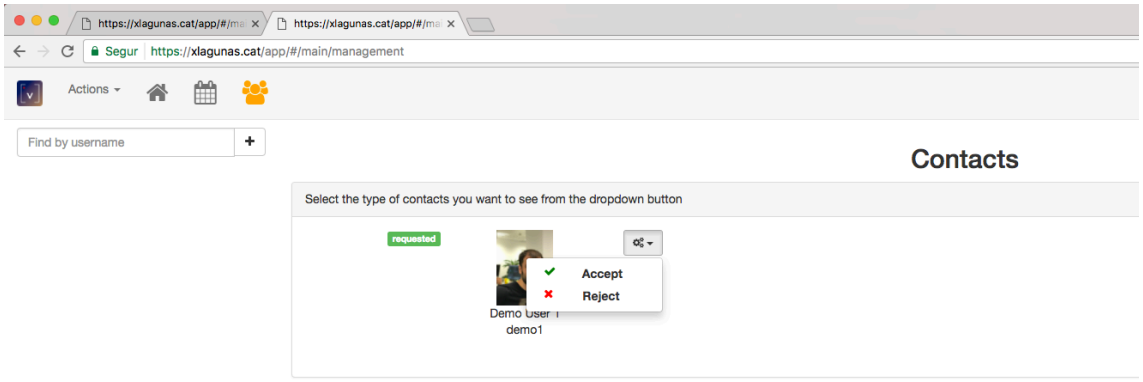

Fig. A.6 Available contact request resolutions

At this point, Demo User 2 can accept or reject the request. If he'd chosen to reject it, Demo User 1 would never notice it because by design, and due to the database being document-based, each user has to have its own relationships with all his contacts so to him, it would appear like User 2 would have never updated the request.

Luckily, Demo User 2 knows the user and he decides to accept him as a contact.

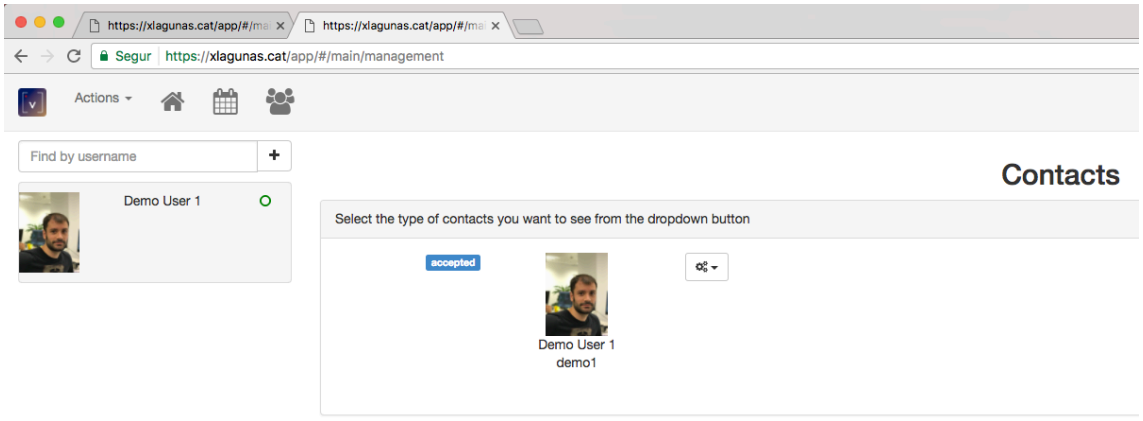

Fig. A.7 Contact added to the roster

Notice how, once the User 2 has accepted the request, it automatically changes to accepted and on top of that, the roster list gets updated with User 1 on it. From that point onwards, both of them can call each other.

At this point, User 2 clicks User 1 on his roster and from there he's shown all the information and options available to interact with. On the image below, can be seen the information provided: username, full name, online status and the actions that can be done: videoconference, voice call or arrange an appointment.

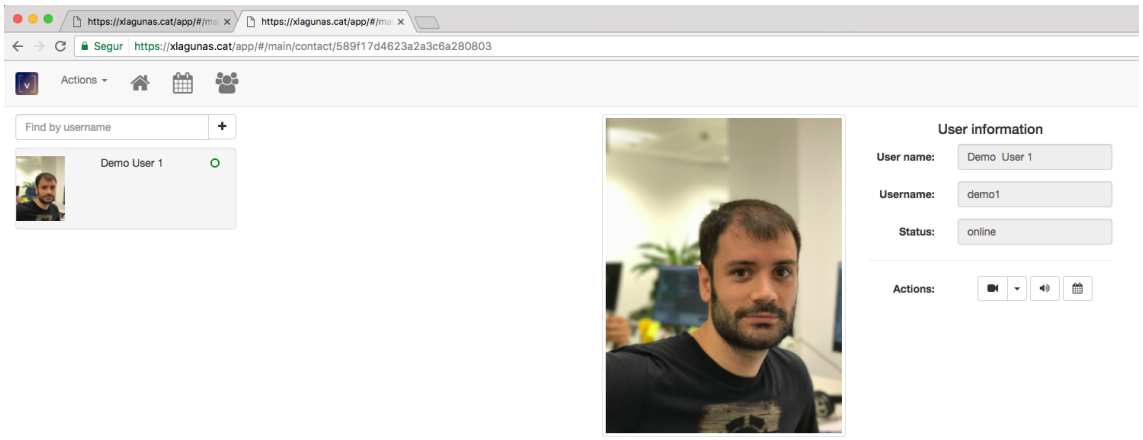

Fig. A.8 Contact details screen

At that point, User 2 choses to start a video call with the user so clicks on the video camera button. On doing so, a confirmation popup appears.

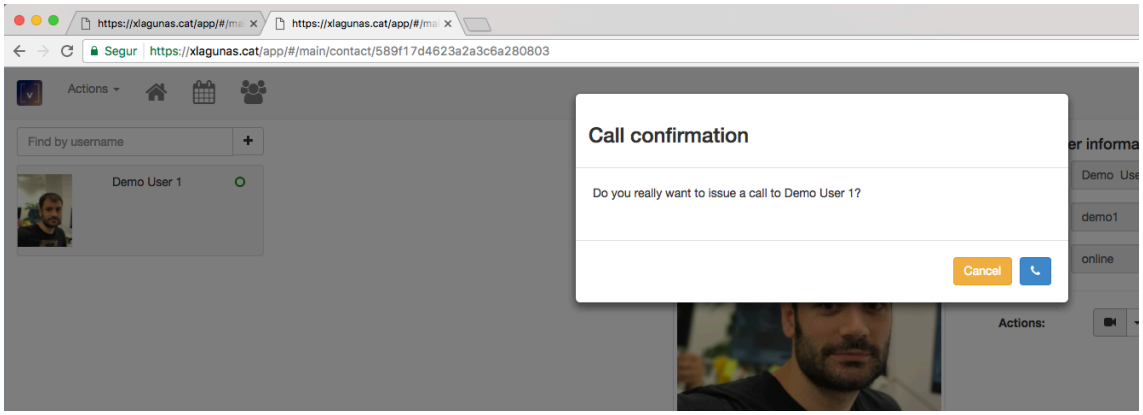

Fig. A.9 Call confirmation dialog

After confirming the call, two different flows spawn at same time, depending on the user. On one side, User 1 will receive the call and a dialog will pop up so he can take action. On the other side, User 2, is shown a completely different popup waiting for the confirmation on the other end.

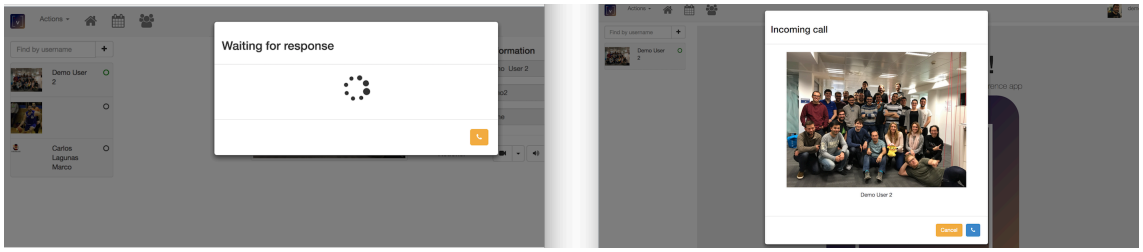

Fig. A.10 dial-in state in both videoconference ends

Finally, after User 1 accepts the call, both users are brought to the conference room where the videoconference will start when both parties will be ready once the signalling process has finished exchanging the information.

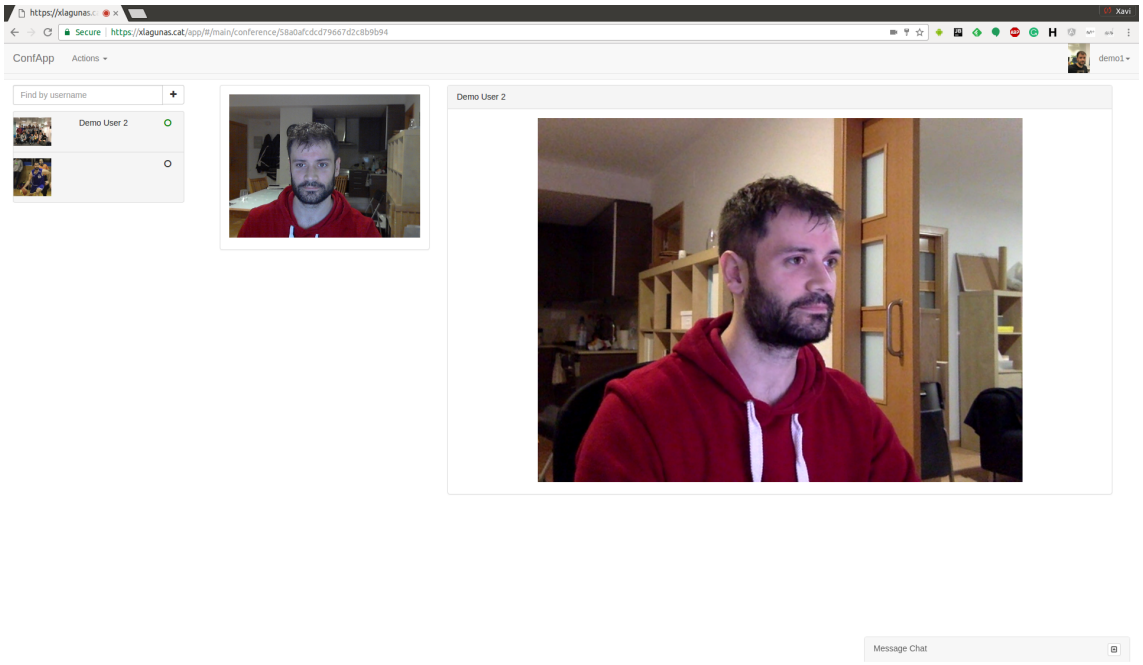

Fig. A.11 videoconference screen

During the videoconference, they make use of the peer-to-peer chat capabilities so that they can securely share private messages.

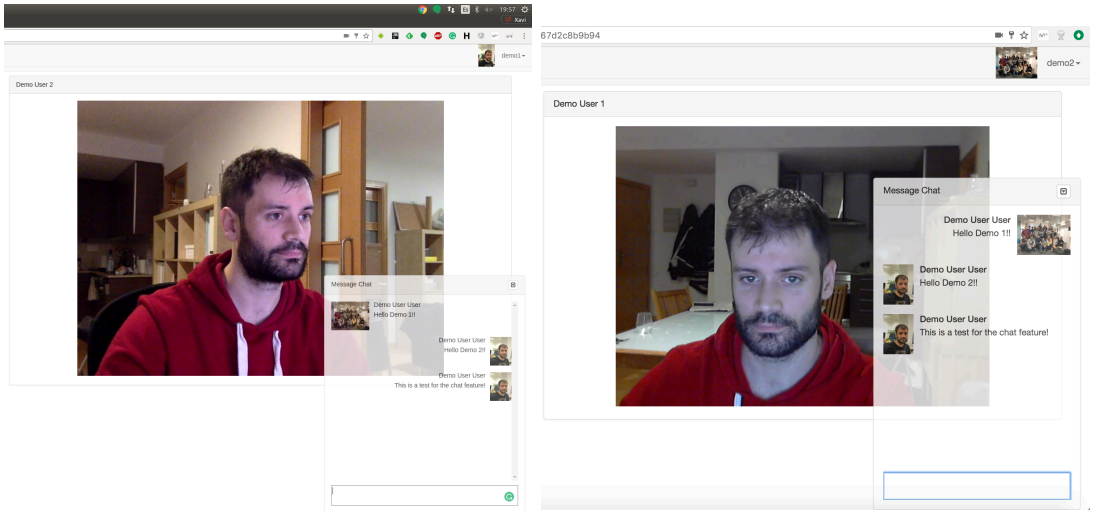

Fig. A.12 WebRTC DataChannel capabilities on chat messaging

Finally after some time discussing, they agreed to talk with User 3, which might have the answer to some of the doubts that they have. Luckily, User 2 notices that the user is available on the phone, so he only needs to drag and drop the user contact to the videoconference and, again, the confirmation call dialog appears. Once the user confirms that he wants to add User 3 to the videoconference, this one goes through the same process described before until he joins the videoconference through his android phone.

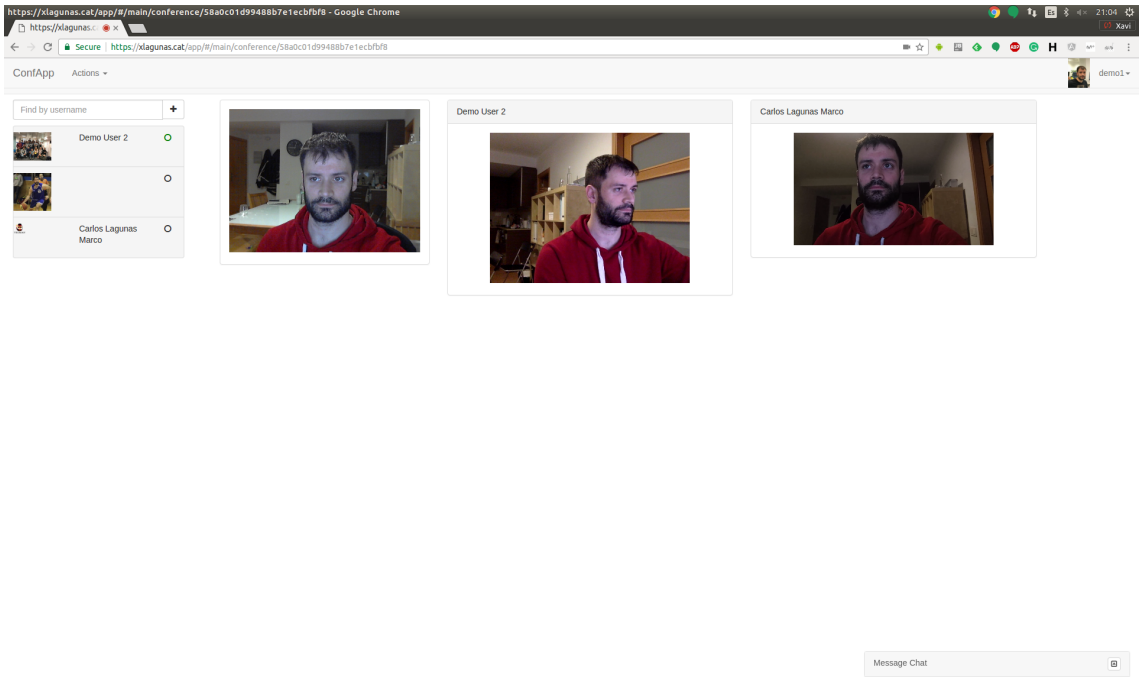

Fig. A.13 Multi-conference between 3 users

On the other hand, user 3, with the native android application sees the same content on the device but with a native interface, which dynamically resizes to show the content of all the users involved.

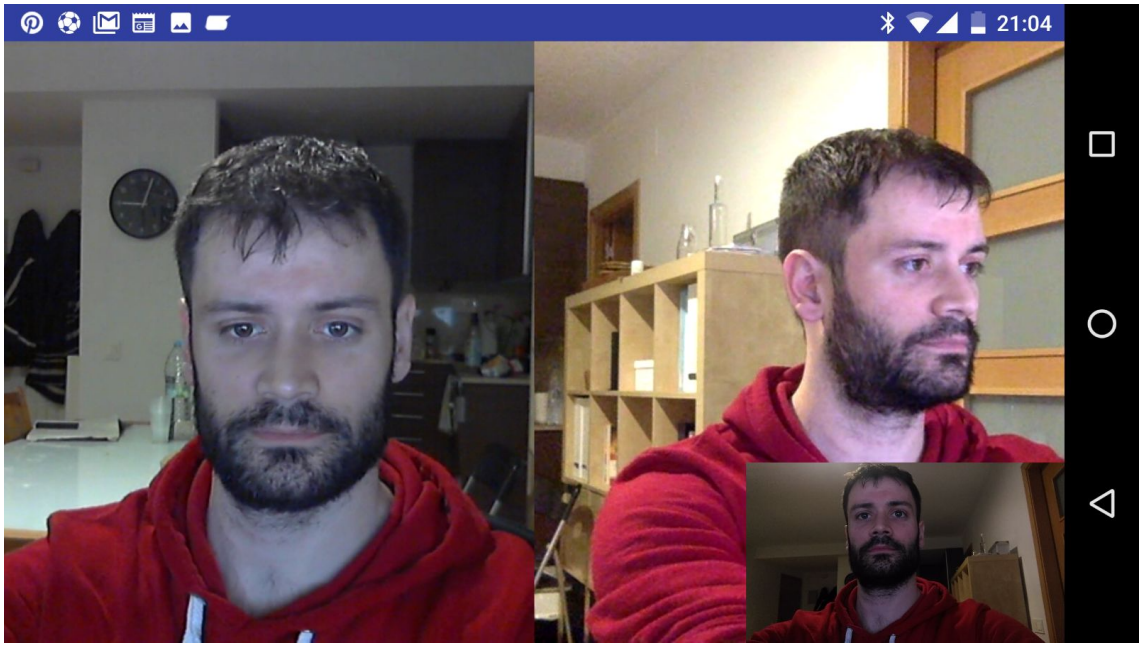

Fig. A.14 Android native application interface